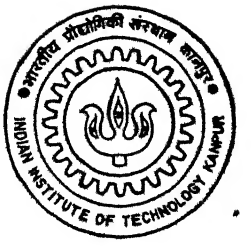
PARALLEL ALGORITHMS FOR SOME COMPUTATIONAL GEOMETRY PROBLEMS

by

A. Pavan Kumar Reddy



DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING

INDIAN INSTITUTE OF TECHNOLOGY KANPUR

FEBRUARY, 1995

Parallel Algorithms for some Computational Geometry Problems

A Thesis Submitted
in Partial Fulfilment of the Requirements
for the Degree of

MASTER OF TECHNOLOGY

by

A. Pavan Kumar Reddy

to the

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING
INDIAN INSTITUTE OF TECHNOLOGY, KANPUR

Feb, 1995

CSE-1995-M-RED-PAR

Acknowledgements

I would like to express my sincere gratitude and heartfelt thanks to my thesis advisor, Dr. Sanjeev Saxena, for extending constant guidance through out this work. He introduced me to the interesting field of Parallel Algorithms. I am grateful to him for patiently explaining all my trivial doubts. I also learnt (at least tried) art of technical paper writing from him.

I thank my examiners and Dr. Tapan K. Sengupta and Dr. Asish Mukhopadhyay for careful reading of thesis and for their constructive suggestions which has led to improved presentation of this thesis. I am also extremely grateful to Dr. Mukhopadhyay for

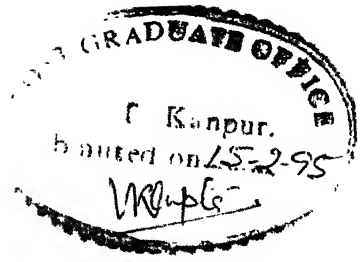
a) Pointing out a special case (and how it should be handled) of one algorithm
b) suggesting possibility of limitation (or error) in another algorithm .
c) drawing attention to ● papers by Goodrich and by Chazelle

I would like to thank Prof. S. Biswas for his excellent teaching. I am grateful to Dr. Mukhopadhyay and Dr. R.K. Ghosh from whom I learnt many things.

I express my sincere thanks to all my friends at I.I.T., Kanpur for the delightful company they have given. I am thankful to Sudershan Banerjee for the useful discussions that I had with him. I thank Srin, RS, Parag and Singhai for helping me in need. As a prominent member of Gaali group I am thankful to my co-members TAN, TRK, Neti and Govind, for making me experience colours of life. I thank Balcha, Kathi, Kraja, Shankar and K(presi) for enriching my cultural life.

I would like to express my deepest thanks to my parents, brother and all my relatives for the constant love that they have shown. I am very grateful to my cousin Pradeep, who has always been a constant source of inspiration for me. Finally I thank Lady Luck for favouring me.

(A. Pavan Kumar Reddy)



Certificate

This is to certify that the work contained in the thesis titled **Parallel algorithms for some computational geometry problems** by **Aduri Pavan Kumar Reddy (Roll No: 9311101)**, has been carried out under my supervision and that this work has not been submitted elsewhere for a degree.

Feb 15, 1995

Dr. Sanjeev Saxena
Assistant Professor,
Department of Computer Science and Engineering
IIT, Kanpur

Abstract

Parallel algorithms for following problems are proposed in this thesis.

1. An optimal $O(\log n)$ time algorithm to find tangents from a set of points to a Convex hull is obtained, provided the point set satisfies some properties.
2. It is shown that shortest path tree in a weak visible polygon can be computed in optimal $O(\log n)$ time with $\frac{n}{\log n}$ processors on a CREW PRAM without triangulating the polygon.
3. An optimal parallel algorithm to triangulate a star shaped polygon is obtained on a CREW PRAM. It works in $O(\log n)$ time with $\frac{n}{\log n}$ processors.
4. A simple parallel algorithm to find convex hull of a simple polygon that runs in $O(\log n)$ time with $\frac{n}{\log n}$ processors is presented. Model of computation is CREW PRAM
5. Some fast parallel algorithms are proposed to compute visibility polygon from a point. First a trivial $O(\alpha(n))$ time algorithm with n^2 processors is described. Then a $O(\log \log n)$ time algorithm with $\frac{n^2}{\log n}$ processors is described. It is shown that this algorithms can be improved to get a algorithm that takes $O((\log \log n)^2)$ time with $\frac{n^{1+\frac{1}{\epsilon}}}{\log n}$ processors where ϵ is a constant (≥ 1). Later it is proved that this algorithm can be easily modified to obtain an optimal $O(\log n)$ time algorithm. All these algorithms run on COMMON CRCW PRAM
6. The reverse problem of computing point set from a Voronoi diagram is considered. A fully optimal parallel algorithm is obtained, on a CREW PRAM.

Contents

Chapter 1

Introduction

Computational geometry is concerned with the computational complexity of geometric problems within the frame work of analysis of algorithms. Before the birth of Computational Geometry, mathematicians studied geometry with the intention of discovering properties of geometric objects. With the emergence of computers many researchers started working in the ares like pattern recognition, computer graphics and image processing where the basic stress was on the design of efficient algorithms based on the geometric properties of the objects involved. As classical treatment of geometric objects was not well suited to the design of efficient algorithms, it became necessary to select the useful concepts that are suitable for the purpose of the field of the algorithms. This marked the birth of discipline "Computational Geometry".

Many problems in this field need to be solved very fast, as these problems come from many real life applications. But we are now at the limits of what can be achieved through sequential computation. So emerged the field "Parallel Computational geometry" to cope up with our needs. As many algorithms designed for sequential computer, are inherently sequential one needs to develop new paradigms suitable for parallel computation.

In this thesis parallel algorithms for some visibility and shortest path problems are studied. These problems have many applications in computer graphics, robotics and pattern recognition. Some of the applications are :

- Suppose a robot is to be moved from one point s to another point t in a polygon. And the robot can not go out of the polygon. The robot should follow the shortest path between the two points with above constraint. This is equivalent to finding the shortest path in a polygon.
- Suppose a communication channel is to be set up between two points in a simple

polygon whose sides are opaque to transmission. The transmission originating at a point s, called source is to be transmitted to a point t, called destination. Clearly, a direct communication is impossible unless the points s and t are in sight of each other inside the polygon. So some repeaters are to be installed in side the polygon to make communication feasible. What is the minimum number of repeaters required. This problem is equivalent to minimum link path between two points in a polygon. Minimum link path in a polygon is the path that has minimum number of vertices.

1.1 Basic definitions

In this thesis, the term "polygon" P, refers to the set of points lying on the boundary and inside it. Boundary of polygon P is denoted by $bd(P)$.

- *Convex hull* of a point set is the smallest convex polygon that includes all points of the set.
- A polygon P is said to *monotone* with respect a line l, if any perpendicular line from l intersects $bd(P)$ at most two places.
- If the edges of a polygon do not intersect except at end points, and each vertex shares exactly two edges then it is called *simple polygon* .
- Two points in a polygon are said to be *visible* from each other if and only if the line segment joining them completely lies inside the polygon.
- A polygon is *star shaped polygon* if there exists a point c, inside it such that all points in P are visible from c (See Figure 1.1a).
- *Kernel* of a polygon is set of all points from which the entire polygon is visible.
- A point p is said to be *weakly visible* from an edge uv of polygon if there exists a point c on uv such that p is visible from c.
- The polygon P is called *weak visible polygon* if all points of P are weakly visible from an edge uv of polygon (See Figure 1.1b).
- *Shortest path* in a polygon is the shortest Euclidian path which does not intersect any edge of polygon.
- *Triangulation* of a polygon P, is division of P into maximum possible number of triangles, such that the vertices of triangles are vertices of P. If P has n vertices it can be divided into $n-2$ triangles.

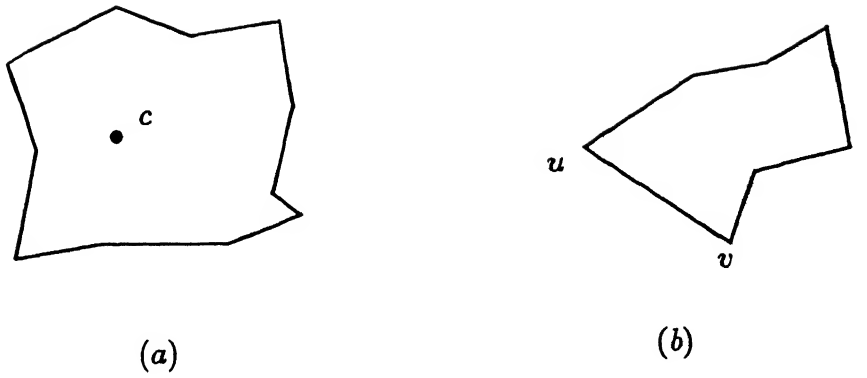


Figure 1.1: a: Star shaped polygon, b: Weak visible polygon

- The expression $right(p_i, p_j, p_k)$ ($left(p_i, p_j, p_k)$) is *true* if the point p_k lies to the right (left) of the ray $p_i p_j$.
- *Voronoi diagram* of a point set S, is the partition of the plane into regions such that each region is the locus of points (x, y) closer to a point of S than any other point of S.

1.2 Parallel computation models

All sequential computers are based on the abstract model Random Access Machine proposed by von-Neumann. It has a read only input tape, on which the program is stored and write only tape on which the machine writes the output of the program, and memory. It stores intermediate values in memory. It can do operations like Addition, Subtraction, Comparison, Jump (from one portion of program to another portion) on the contents of its memory. It can also read from input tape and write on output tape. The flow of the program in RAM model is sequential, i.e it can not execute more than one instruction at a time.

A Parallel Random Access Machine employs p synchronous RAMs, all having unit time access to a shared memory. All the RAMs share the common program stored in read only tape. At any point of time the RAMs can either read from or write in the common memory or execute a step of the program. There are different types of PRAMs which differ in their capacity to do simultaneous reads and writes. From now we call a RAM as processor.

An Exclusive Read Exclusive Write (EREW) PRAM allows neither simultaneous read nor simultaneous write by more than one processor to the same memory location.

A Concurrent Read Exclusive Write (CREW) PRAM allows simultaneous reads but

not writes.

A Concurrent Read Concurrent Write (CRCW) COMMON PRAM allows both simultaneous reads and writes to same location by more than one processor, but a write attempt is valid only if all the processors try to write the same value in that location.

If the worst-case running time of the best known sequential algorithm for a problem is t, an optimal parallel algorithm for the same problem runs in $\frac{t}{p}$ time using p processors. An optimal fully parallel algorithm for the same problem runs in constant time with t processors.

1.3 Overview of thesis

In Chapter 2 the problem of finding tangents from a set of points to a convex hull is considered, when the point set has a special property. This problem seems to have many applications. We use this as sub-step in algorithms presented in Chapter 3 and Chapter 4. An optimal $O(\log n)$ time algorithm with linear work is proposed for this problem. In Chapter 3 an optimal algorithm which uses $\frac{n}{\log n}$ processors and takes $O(\log n)$ time to computed shortest path tree in a weak visible polygon is described. An optimal $O(logn)$ time algorithm for triangulating simple polygon is presented in Chapter 4. A simple parallel algorithm to find the convex hull of a simple polygon using the concepts of visibility is obtained in chapter 5. All algorithms in these chapters work on CREW PRAM. In chapter 6 some fast parallel algorithms to find the visibility polygon from a point are presented. First a trivial $O(\alpha(n))$ time algorithm is described. Then an $O(\log\log n)$ time algorithm with $\frac{n^2}{\log n}$ processors is discussed. It is also shown that visibility polygon from a point can be computed in $O((\log\log n)^2)$ time with $\frac{n^{1+\frac{1}{c}}}{\log n}$ processors. With a slight modification of this algorithm an optimal $O(\log n)$ time algorithm for the same problem is obtained. In chapter 7 a fully parallel algorithm for finding the point set from a Voronoi diagram is described.

Chapter 2

Finding Tangents from a Set of Points to a Convex Hull

2.1 Introduction

In this chapter we describe a problem which will be used as a sub-step in Chapter 3 and Chapter 4. It is known that tangent to a convex hull from a point can be easily computed in $O(\log n)$ time. In fact tangent between two convex hulls can also be computed in $(\log n)$ time [29]. Here we consider the problem of finding tangents from all vertices of a polygon to a convex hull, where the vertices of the polygon have a special property. Assume that vertices of convex hull C are indexed $c_1, c_2, \cdots, c_n$ in clockwise direction. Also assume that the vertices of P are indexed $p_1, p_2, \cdots, p_m$ in anti-clockwise direction. Chain(p_i, p_j) denotes the boundary of P formed by the vertices $p_1, p_2, \cdots, p_m$ in anti-clockwise order. Chain(c_i, c_j) is also defined similarly. Let the boundary of polygon be denoted by $bd(P)$. In this chapter we first describe a sequential algorithms that runs in $O(m+n)$ time, then an optimal parallel algorithm which takes $O(\log n)$ time is proposed.

2.2 The algorithm

The problem of finding tangents to a convex hull C from an arbitrary point set with linear cost is open. However, it is possible to obtain a linear time algorithm, if the vertices of the polygon satisfy following property.

Property 2.1 If tangent from a point p_i to C touches C at c_m then tangent from any point p_j where $j > i$ touches C at a point c_n where $n \leq m$.

First consider sequential version of the algorithm.

1. Find the tangent from p_1 to C. Let it touches C at c_i.

2.0. Assume that we have proceeded up to p_i and we have to find tangent from p_{i+1}. Let the tangent from p_i touches C at c_j.

2.1. If right(c_{j+1}, c_j, p_{i+1}),then $p_{i+1}c_j$ is the tangent i=i+1, GoTo 2.

2.2. If left(c_{j+1}, c_j, p_{i+1}) then j=j+1, GoTo 2. 1

It can be seen that the above algorithm works in $O(m+n)$ time

Now consider the parallel version of the algorithm.

1. Divide $bd(P)$ into $\frac{m}{\log n}$ parts each having $\log n$ points. Call first point in each part tail and last point head.
2. Find tangents to C from head and tail of each part.
3. for all parts pardo

3.1 Consider each part as Chain(p_i, p_j) with p_i as tail and p_j as head. Let tangents from p_i and p_j touch the convex hull C at c_i and c_j respectively. Divide the chain chain(c_i, c_j) into $\frac{car(i,j)}{\log n}$ parts, where $car(i,j)$ is number of points in Chain(c_i, c_j). Define head and tail similarly for each part of C.

3.2 For each head in Chain(c_i, c_j) find the point p_k in Chain(p_i, p_j) such that the tangent from head passes through p_k.

REMARK: Thus, we have divided P into $\frac{m}{\log n}$ parts, each part having at most $\log n$ points. The tangents from heads of these parts divide C into $\frac{m}{\log n}$ parts but each part may have any number of points. Step 3.1 divides C into $\frac{m}{\log n}$ parts such that each part has at most $\log n$ points. Step 3.2 divides further divides each part of $bd(P)$ such that each subpart has at most $\log n$ points, and overall P is divided into at most $\frac{(m+n)}{\log n}$ parts. C is also divided into at most $\frac{(m+n)}{\log n}$ parts.

3.3 Take corresponding parts in P and C and apply sequential algorithm to each part.

Claim 2.1 Above algorithm works in $O(\log n)$ time with $\frac{(m+n)}{\log n}$ processors.

Proof: Step 2 can be done in $O(\log n)$ time with $m/\log n$ processors. Step 3. 2 can be done in $O(\log n)$ time with $\frac{m+n}{\log n}$ processors. In this step for a point $c_i \in C$ we have to find a point from a group of at most $\log n$ points, which can be done in $O(\log n)$ time with a single processor. So over all algorithm works with above processor and time bounds. ∎

Chapter 3

Shortest Path Tree in a Weak Visible Polygon

3.1 Introduction

Finding shortest paths in a polygon is one of the basic problems in Computational Geometry. Shortest path problems arise in many application areas such as computer graphics, robotics and vision. Routine for shortest path is also used in other computational problems like, finding the visibility information [16], and minimum link path in a polygon[34]. Therefore, a great deal of research work has been devoted to finding efficient algorithms for shortest path and its variants [16, 34] (like link paths and L_1 paths).

Lee and Preparata [26] describe a linear time algorithm to find shortest path between two vertices in a simple polygon. Guibas *et al.*[23] has shown that the shortest path tree rooted at any point in a polygon can be computed in linear time. Both algorithms assume that polygon is triangulated. As Chazelle [11] recently proposed a linear time algorithm for triangulating a simple polygon we can sequentially compute shortest path tree rooted at any vertex in linear time. Ghosh *et al.* [18] describe a linear time algorithm for finding shortest path tree in a weak visible polygon, without triangulating the polygon. They solve the problem with a new characterisation of weak visible polygon in terms of shortest path between two arbitrary vertices of the polygon.

Attempts have also been made to obtain parallel algorithms for these problems. Goodrich *et al.* [22] obtain a parallel algorithm for computing the shortest path tree in

$O(\log n)$ time with $\frac{n}{\log n}$ processors(on CREW model), if the polygon is triangulated. But the best known parallel algorithm for triangulating a polygon takes $O(\log n)$ time with n processors [21]. As an optimal algorithm for triangulating weak visibility polygon is not known, we cannot use the algorithm of Goodrich *et al.* to get an optimal algorithm even if the polygon is weak visible. Here we propose an optimal $O(\log n)$ time algorithm for finding shortest path tree in a weak visible polygon. Our model of computation is Concurrent Read Exclusive Write (CREW). We use the characterisation of Ghosh *et al.* to solve our problem. The algorithm uses rootish divide and conquer paradigm [5, 20] to reduce resource bounds.

We assume that the weak visible polygon P is given as (array of) counterclockwise sequence of vertices (i.e., their respective x and y coordinates) $p_1, p_2, \cdots, p_n$. Assume that the polygon is weakly visible from the edge p_1p_2. The symbol P denotes both the boundary of the polygon (bd(P)) as well as the region bounded by it. Chain(p_i, p_j) denotes the boundary formed by the vertices $p_i, p_{i+1}, \cdots, p_j$ in counter clockwise order. The shortest path between two vertices of polygon should not intersect any edge of the polygon. The shortest path between two vertices p_i and p_j is denoted as $SP(p_i, p_j)$. Given any three points $p_i = (x_i, y_i), p_j = (x_j, y_j)$ and $p_k = (x_k, y_k)$, let $S = x_k(y_i - y_j) + y_k(x_j - x_i) + y_jx_i - y_ix_j$. If $S < 0$ then we say p_k lies to *right* of the line passing through p_i and p_j. If $S > 0$, then it lies to the *left* of the same line. If the interior angle at a vertex p_i is convex then v_i is called a *convex vertex*. An edge p_ip_{i+1} is called *convex edge* if both p_i and p_{i+1} are convex vertices. Two vertices are *visible* from each if and only if the segment joining them lies completely inside P. A point p is said to be *weakly visible* from an edge st, if there is a point z on the line segment st such that p and z are visible. If every point in P is weakly visible from st then P is said to be *weakly visible* from st and P is called a *weak visible polygon.*

3.2 Properties of weak visible polygons

We assume that the edge p_1p_2 is convex edge. If a vertex p_i of P is weakly visible from convex edge p_1p_2 of P, then the following properties hold [23]

Property 3.1 1.1 *$SP(p_2, p_i)$ makes right turn at every vertex in the path*

1.2 *$SP(p_1, p_i)$ makes left turn at every vertex in the path*

1.3 *$SP(p_1, p_i)$ and $SP(p_2, p_i)$ are two disjoint paths and they meet only at p_i*

1.4 *The region enclosed by* $SP(p_1, p_i)$ *and* $SP(p_2, p_i)$ *and* p_1p_2 *is totally contained inside* P.

Ghosh *et al.* [18] prove following additional properties of weak visible polygon.

Property 3.2 2.1 *For any two vertices* p_i *and* p_j, *no vertex, except those of* $Chain(p_i, p_j)$ *can be on* $SP(p_i, p_j)$.

2.2 *For any two vertices* p_i *and* p_j, $SP(p_i, p_j)$ *makes right turn at every vertex in the path.*

Thus from property 3.2, for any two vertices p_i and p_j, $\mathrm{SP}(p_i, p_j)$ is the same as the convex hull of $\mathrm{Chain}(p_i, p_j)$.

Let T_i be the shortest path tree of polygon chain- $\mathrm{Chain}(p_r, p_{r+m})$, rooted at vertex p_r (for any vertex p_r). Further, let p_i be any internal node in this tree, with children $p_i^1, p_i^2, \cdots, p_i^f$. We define an *ordering* on these children as follows: draw a straight line from p_i to any child p_i^k. If x is the number of children of p_i lying to right of the line $p_ip_i^k$ then we define *rank* of p_i^k to be x. We also define an ordering among leaf nodes. Rank of a leaf node l_i is zero if every node in the path from p_r to l_i has a rank zero. Let l_i and l_j be any two leaf nodes, with lowest common ancestor, $L = LCA(l_i, l_j)$. Let us assume that the rank of first node in the path from L to l_j is *greater than* the rank of first node in the path from L to l_i. Then rank of l_i and l_j are defined in such a manner that rank of l_j is greater than rank of l_i; note that ranks are in range 0 to number of leaves in tree -1.

If p_i is a leaf node with rank f then $nextleaf(p_i)$ will denote the leaf node with rank $f + 1$. If p_i is an internal node then $SuccT(p_i)$ will denote the child of p_i which has rank zero. For a leaf node p_i, $SuccT(p_i)$ is the first node in the path from lowest common ancestor of: p_i and $nextleaf(p_i)$, to $nextleaf(p_i)$. And $Succ(p_i) =$ the successor of p_i in P $(= p_{i+1})$ for all nodes. In T_i the leaf with rank zero is called the *left most leaf*, and the leaf with maximum rank is called the *right most leaf*.

Lemma 3.1 For any node p_i in tree T_i, $SuccT(p_i) = Succ(p_i)$.

Proof: First consider the case when p_i is not a leaf node. We prove the Lemma by induction on the level of node p_i.

Base case: The node p_i is at level zero, i.e it is root p_r. By Property 3.2, shortest path from p_i to p_{i+1} $(= Succ(p_i))$ should be the edge p_ip_{i+1}. Thus $Succ(p_i)$ is at level one. Assume that $Succ(p_i)$ is not $SuccT(p_i)$, but some other vertex p_j. Note that there is an edge $p_{r-1}p_r$ in the polygon. Now there should be a polygon chain from p_j to p_{r-1} (to form the polygon P), and the tree edge $p_iSuccT(p_i)$ should lie inside P.

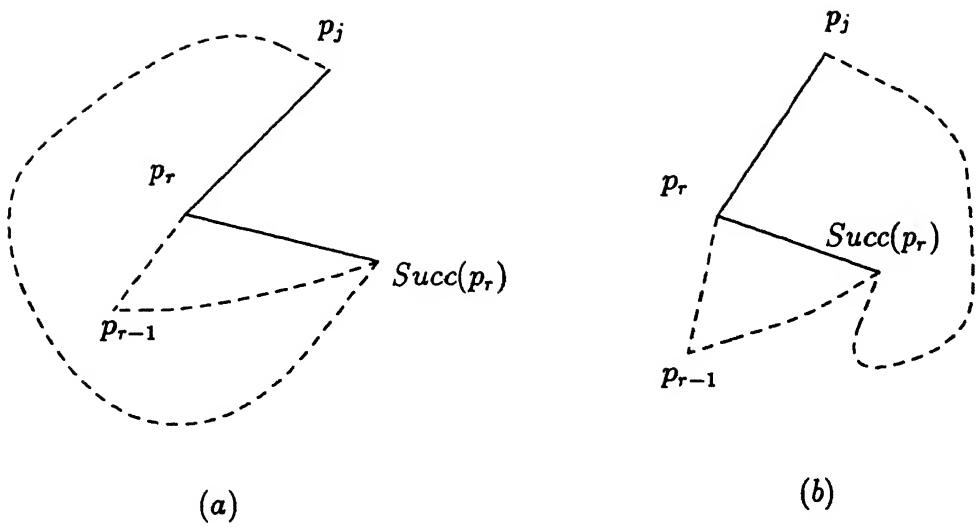


Figure 3.1: Proof of Lemma 3.1

Further it should be possible to reach p_{r-1} from p_j along this chain during anti-clock wise traversal. Observe that the chain from p_j to p_{r-1} can be formed in two different ways (Figure 3.1). In the first case, the tree edge $p_iSuccT(p_i)$ lies outside P; and $p_isuccT(p_i)$ can not be an edge of shortest path tree, a contradiction.
In the second case the tree edge $p_iSuccT(p_i)$ lies inside P, but in this case we are doing a clockwise traversal to reach p_{r-1}, which means that the polygon vertices are indexed in clockwise order, this contradicts our basic assumption that P is given as anti-clockwise sequence of vertices. Hence $SuccT(p_i) = Succ(p_i)$ when p_i is root node. This completes the base case.

Induction argument: Let the claim be true for all nodes up to level h. Let p_i be any node at level at $h+1$. Assume that $Succ(p_i)$ is not $SuccT(p_i)$, but some other vertex p_j. This implies that the vertices in path from p_r to p_i and then to p_j form a polygon chain. So shortest path from p_r to p_j should pass through the vertices of this chain only. But it is passing through some other vertices (vertices in the tree path from p_r to p_j); again a contradiction. So p_j cannot be $Succ(p_i)$, and $Succ(p_i) = SuccT(p_i)$.

The proof when p_i is a leaf node is similar to that of base case. ∎

Corollary 3.1 In a tree T the vertex with maximum index is the right most leaf. ∎

Note that if we know the shortest path from p_i to p_j and also the shortest path from p_j to p_k, where both are convex paths, then to find the shortest path from p_i to p_k we have to merge the two hulls by finding the tangent.

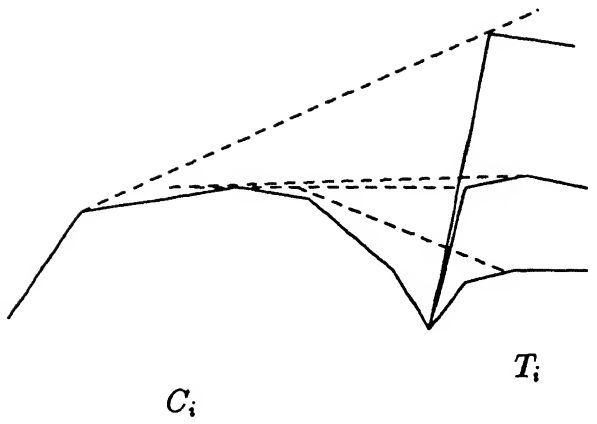


Figure 3.2: The sub-problem, Finding tangents to C_i

Finding shortest path tree of the given polygon requires solution of following intermediate problem:

Problem 3.1 Given a convex hull C_i, and a layer of hulls (here layer of hulls is the tree T_i) find tangents from each point in the layer of hulls to C_i.(see Figure 3.2).

We solve the Problem 3.1, using $\frac{m}{\log n}$ processors in $\log n$ time, where m is total number of points, in section 3.3.

Lemma 3.2 Assume that the polygon P is divided into k polygon chains, of consecutive vertices. Suppose that shortest path tree T_i of each of chain i is known $(i = 1, \cdots, k)$, and that the right most leaf of tree T_{i-1} is the same as the root of tree T_i. Assume that root of first tree T_1 is p_r. Then shortest path tree of the polygon P rooted at p_r can be obtained by merging trees $T_1, T_2, \cdots, T_k$ in $O(\log n)$ time, with $\max(k^2, \frac{n}{\log n})$ processors on a CREW PRAM; here n is the total number of vertices in polygon P.

Proof: Let r_i be the root of T_i. Observe that we can look each tree T_i as layers of hulls (see Figure 3.3). First we will merge the upper most hulls of all the trees (i.e. first we will find the shortest path from p_r to some vertices that lie on the upper hull of each tree.); for this we may consider only the upper most hull and ignore all other vertices in the tree. Assign one processor to each pair (T_iT_j) for $j < i$ and find tangents from T_i to all T_js, where $j < i$. Consider the set of tangents to T_i. Divide these tangents into two sets. First set consists of tangents that pass through some vertex p_i (of T_i). Second set consists of tangents that pass through the vertices p_j where $j < i$. Discard all the tangents of second set. Of all the tangents in first set keep the tangent which makes maximum angle with x-axis. Discard other tangents

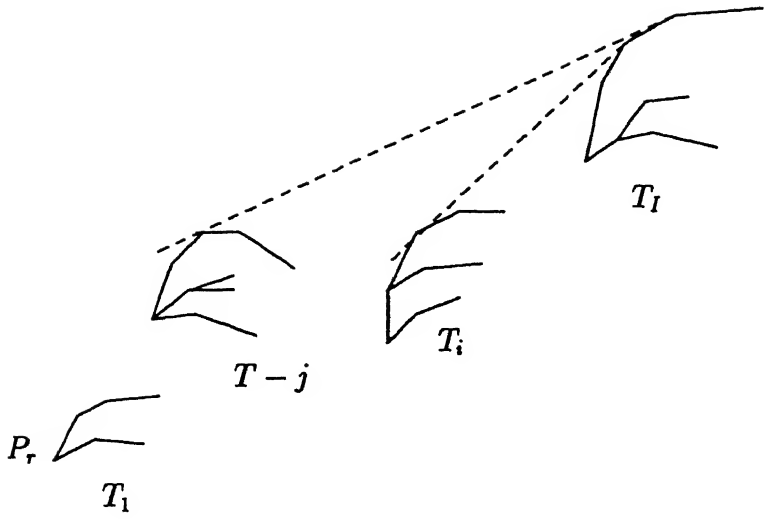


Figure 3.3: Merging shortest path trees

to T_i. This gives a partial shortest path tree. Now we have to find shortest path from P_r to the vertices in the lower hulls (of T_i) and to the vertices that lie between r_i and p_i on the upper most hull, of T_i (for all trees). Consider C_i, the shortest path from p_r to r_{i-1} where r_{i-1} is root of T_i (or right most leaf of T_{i-1}). To find shortest path p_r to all other vertices of T_i, it is sufficient to find tangents from all these vertices of T_i to C_i. As we can solve Problem 3.1 in same resource bounds, we are done. ∎

REMARK: By Corollary 3.1, the right most leaf of tree T_{i-1} is the same as the root of tree T_i.

Thus, we can find the combined shortest path tree of the k-chains in $O(\log n)$ time with $max(k^2, m/\log n)$ processors, where m is total number points in all the trees. The algorithm to compute shortest path tree in P is:

1. If the problem size is less than $\log n$ then solve sequentially, else, divide the polygon chain into $\sqrt{\frac{n}{\log n}}$ parts, each part having at most $\sqrt{n \log n}$ points.
2. Solve each sub-problem recursively
3. Merge all $\sqrt{\frac{n}{\log n}}$ sub-problems, as described in Lemma 3.2

Claim 3.1 Above algorithm works in $O(\log n)$ time with $\frac{n}{\log n}$ processors.

Proof: By Lemma 3.2, the sub-problems can be merged in $O(\log n)$ time. The recurrence equations are

$T(\log n) = \log n$

$T(n) = T(\sqrt{n \log n}) + \log n$

Solving the recurrence the equation we get $T(n) = O(\log n)$. To merge k sub-problems we need $\max(k^2, m/\log n)$ processors (Lemma 3.2). As k is always less

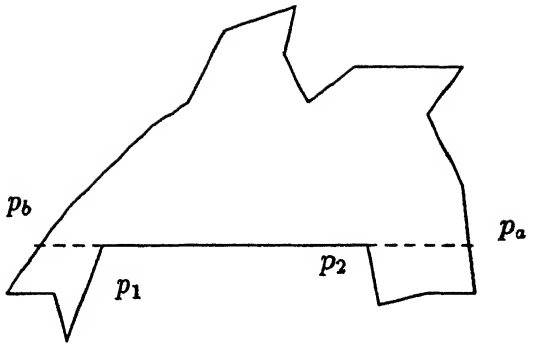


Figure 3.4: p_1p_2 is not convex edge

than $\sqrt{n \log n}$ and m is less than n, $\frac{n}{\log n}$ processors are sufficient. ∎

So far we have assumed that edge p_1p_2 is convex. Let us relax this condition. If p_1p_2 is not convex, then extend the edge p_1p_2 so that it intersects $bd(p)$ at p_a and p_b as shown in Figure 3.4. Delete chains Chain(p_2, p_a) and Chain(p_b, p_1) and find the Shortest path tree in the new polygon. We have to find shortest paths to the vertices in the chains, Chain(p_2, p_a) and Chain(p_b, p_1). Note that the polygon formed by Chain(p_2, p_a) and the edge p_ap_2 forms a star shaped polygon with p_2 as a star point. Similarly the polygon formed by the chain Chain(p_b, p_1) and the edge p_1p_b forms a star shaped polygon with p_1 as star point. We can easily triangulate these star shaped polygons as the star point is one of the vertices and find shortest path tree in these polygons [22]; basically, we will be done if we join vertex p_2 with all other vertices in the polygon.

Although, the algorithm as described, computes the shortest path tree rooted at p_2, we can easily modify the algorithm to compute the shortest path tree rooted at any vertex. Let us assume that we have to compute the shortest path tree rooted at vertex p_i. Consider all vertices that lie in the chain, Chain(p_i, p_1) and find the shortest path tree rooted at p_i for these vertices as above. To find the shortest paths for the vertices that lie in the chain Chain(p_2, p_i), reverse the order of vertices so that now they are arranged in clockwise order. Use the same algorithm, except that roles of "left" and "right" are interchanged. Thus we have the following Theorem:

Theorem 3.1 In a weak visible polygon the shortest path tree rooted at any vertex can be found in $O(\log n)$ time with $\frac{n}{\log n}$ processors.

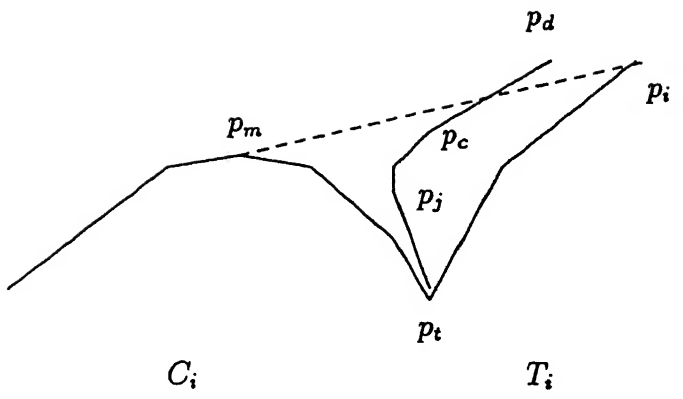


Figure 3.5: Proof of Property 3.3

3.3 Solution of Problem 3.1

In this section we consider the solution of Problem 3.1. We are given a convex hull C_i, and a layer of hulls (here layer of hulls is the tree T_i) and are required to find tangents from each point in the layer of hulls to C_i (see Figure 3.2).

Let us assume that tree T_i has m hulls (i.e T_i has m leaves). We will divide the points in all layers (of hulls) into two sets. If p_t is top most point of a hull (of T_i) then we put the points from r_i to p_t in the first set, and rest of the points of the hull in the second set. We describe the algorithm to find tangents from points in the first set; tangents from points in the second set can be similarly found. We do not know of any linear time sequential algorithm for finding tangents from an arbitrary set of points to a convex hull. However the points in the problem under consideration are vertices of a weak visible polygon, using Property 3.3, it is possible to find tangents with linear cost.

Property 3.3 If tangent from a point p_i to C_i touches C_i at p_m then tangent from any point p_j where $j > i$ should touch C_i at a point p_n where $n \leq m$.

Proof: Let the tangent from p_i to C_i touch C_i at p_m. Let p_j $(j > i)$ be any other point. Note that tangent from p_j can pass through point p_k $(k > m)$ (see Figure 3.5) only if p_j lies to left of tangent $p_i p_m$. In this case it can be easily observed that there exists an edge $p_c p_d$ ($c, d > i$) which intersects the tangent $p_i p_m$ (Figure 3.5). This implies that shortest path from p_r to p_i is passing through some other vertex which does not lie in the polygon chain Chain(p_r, p_i), which contradicts Property 3.2 of weak visible polygon. ∎

Note that this property is the same as Property 2.1 of Chapter 2. Hence Prob-

lem 3.1 can be solved optimally. This completes the proof of Theorem 3.1.

Chapter 4

Triangulation of Star Shaped Polygons

4.1 Introduction

Algorithm for triangulating simple polygons finds applications in many other problems in Computation Geometry. The algorithm for triangulation is used as a preprocessing step in algorithms like planar point location [24, 27, 29]. Triangulation algorithm is also used in finding visibility information and computing shortest paths [16, 26]. Clearly, these problems can be solved more efficiently, if the input is given as a triangulated polygon. On sequential computational model, many algorithms are available to triangulate a simple polygon. Garey *et al.* obtain an $O(n \log n)$ time algorithm for this problem. They solve the problem by dividing the polygon into monotone polygons, and then triangulate them. Fournier and Montuno [15] also describe an algorithm with same time complexity; here, the polygon is divided into trapezoids and then into one sided monotone polygons. Tarjan and Van Wyk [35] first break the $O(n \log n)$ bound by obtaining an $O(n \log \log n)$ time algorithm to triangulate a polygon. They also obtain a linear time algorithm if the polygon is weakly visible from an edge. Recently Chazelle [11] has obtained a linear time algorithm to triangulate a simple polygon, thus solving one of the well known open problems in computational geometry. If the polygon has special structure then it possible to obtain simpler algorithms (with smaller constant factors); extremely simple linear time algorithms exist for triangulation, when the polygon is monotone or star shaped [15].

In parallel setting, Goodrich [21] obtains an $O(\log n)$ time algorithm with n pro-

cessors to triangulate a polygon, on a CREW PRAM. The algorithm works even when polygon has holes. If the trapezoidal decomposition of polygon is given then the algorithm takes, $O(\log n)$ time with $\frac{n}{\log n}$ processors. Yap [38] has also obtained another algorithm with same resource bounds. If the polygon is monotone then the problem can be solved optimally in $O(\log \log n)$ time [8] on a CRCW PRAM. In this chapter we propose an optimal algorithm for triangulation, if the polygon is star shaped. Our model of computation is the CREW PRAM.

We assume that the star shaped polygon P is given as (array of) counterclockwise sequence of vertices (i.e., their respective x and y coordinates) $p_1, p_2, \cdots, p_n$; $x(p_i)$ and $y(p_i)$ denote x and y co-ordinates of the point p_i. Assume symbol P denotes both the boundary of the polygon ($bd(P)$) as well as the region bounded by it. Chain(p_i, p_j) denotes the boundary formed by the vertices $p_i, p_{i+1}, \cdots, p_j$ in counterclockwise order. CH(p_i, p_j) stands for the convex hull of Chain(p_i, p_j). Two vertices are *visible* from each other if and only if the segment joining them lies completely inside P. The polygon P is called *star shaped* if there exist a point c inside P such that the entire polygon P is visible from c; c is called *kernel* of the polygon. If a horizontal line is drawn through a vertex p_i, and if adjacent edges of p_i lie on both sides of the horizontal line then p_i is called a *regular vertex* . The vertex p_i is called *stalagmitic vertex* if both adjacent edges lie below the horizontal line, and vertex p_i is called *stalactitic vertex* if both adjacent edges lie above the horizontal line. We call a vertex p_i *bottom peak vertex* if it is a stalactitic vertex and it lies below the horizontal line through the kernel point c (these definitions are from [15]). We call *top peak vertex* if it lies above the kernel point c and it is a stalagmitic vertex.

4.2 The algorithm

We triangulate a star shaped polygon by decomposing the polygon into smaller and more simpler objects like monotone polygons.

Without loss of generality, assume that the left most point of the polygon is p_1. Draw a horizontal through the kernel point c. Let it intersects the edge $p_{a-1}p_a$ on the left side and the edge $p_{b-1}p_b$ on the right side (note that the line should intersect the polygon at only two points as P is star shaped). We call Chain(p_a, p_b) the *lower chain* and the Chain(p_b, p_a) the *upper chain* (see Figure 4.1).

Fact 4.1 If p_i and p_j are any two vertices of lower chain such that $i < j$ then the

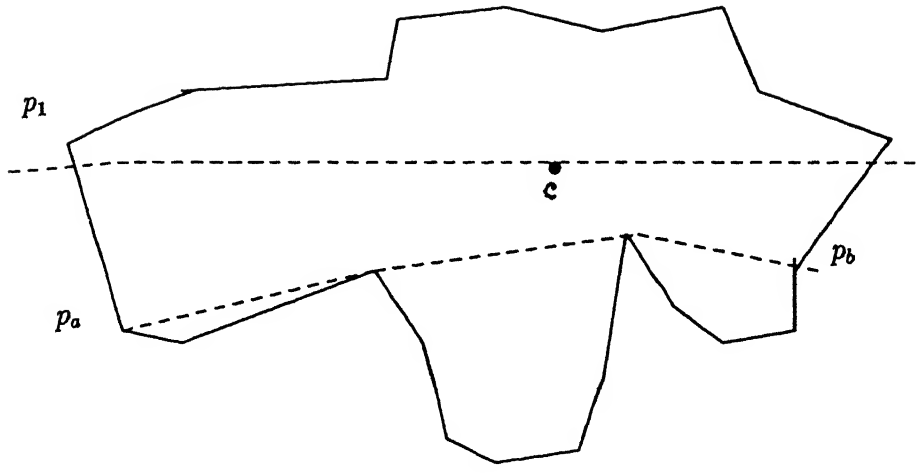


Figure 4.1: The star shaped polygon P, and lower hull

convex hull of the vertices that lie in Chain(p_i, p_j), (CH(p_i, p_j)) does not intersect any other part of the polygon P.

Proof: The kernel point c can at most touch the CH(p_i, p_j) but cannot lie inside it as Chain(p_i, p_j) is part of the lower chain. Now assume that there is some edge $(p_l p_m)$ of the polygon P that intersects an edge of CH(p_i, p_j). This means that one of the vertices p_l or p_m is not visible from c, a contradiction. ∎

Consider the bottom peaks of the lower chain, let p_{lb} be the bottom peak with minimum index. Let p_{lt} be the top peak with maximum index in upper chain. Then observe that

Fact 4.2 The chain, Chain(p_{lt}, p_{lb}) is monotone with respect to y axis.

Proof: All vertices in the chain, Chain(p_{lt}, p_{lb}) are regular vertices. This is so because if a vertex is not regular then it should either be a stalagmitic vertex or a stalactitic vertex; this implies that the vertex will be either a top peak, or a bottom peak.

By definition of regular vertex, if p_i is a regular vertex then $y(p_{i-1}) < y(p_i) < y(p_{i+1})$. Thus, the Chain$(p_{lt}, p_{lb})$ is monotone with respect to y axis. ∎

Similarly we can prove that Chain(p_{rb}, p_{rt}) is also monotone with respect to y axis, where p_{rb} is the vertex with maximum index among bottom peaks and p_{rt} is the vertex with minimum index among top peaks.

We next describe the algorithm.

Step 1 Find upper and lower chains, by finding vertices p_a and p_b.

REMARK: This can be done in $O(\log n)$ time.

Step 2 Find the hull of lower chain (portion lying between p_a and p_b). The hull divides polygon P into two parts the upper part (P_u) and the lower part (P_l). We will consider the problem of triangulating P_l; P_u can be triangulated in a similar manner.

Observe that if the portion of hull between p_a and p_b has k edges, then polygon P_l is in turn divided into k sub polygons $(P_l^1, \cdots, P_l^k)$. Each sub polygon is bounded by the hull edge and the polygon chain lying below that edge. If the polygon P_l^i has x_i points then assign $\frac{x_i}{\log n}$ processors to it.

REMARK: As in this step we are finding the convex hull of a simple polygon it can be implemented in $O(\log n)$ time with $\frac{n}{\log n}$ processors [37]

Step 3 For all polygons $p_l^1, \cdots, p_l^k$ do

Step 3(a) Identify all non-regular vertices in P_l^i and find non-regular vertices having the maximum (p_r) and minimum index (p_l). Also find the vertex p_f having the minimum index among all vertices of P_l^i and the vertex p_e having the maximum index among all points of P_l^i.

Step 3(b) Find following convex hulls— CH(p_f, p_l), the convex hull of Chain(p_f, p_l); CH(p_l, p_r) the convex hull of Chain(p_l, p_r) and CH(p_r, p_e) the convex hull of Chain(p_r, p_e).

Step 3(c) Triangulate following polygons—

the polygon bounded by chains Chain(p_l, p_r) and CH(p_l, p_r); details of triangulation algorithm will be described later

the *monotone* polygon bounded by chains Chain(p_f, p_l) and CH(p_f, p_l) (see Fact 4.3 below) and

the *monotone* polygon bounded by chains Chain(p_r, p_e) and CH(p_r, p_e) (see Fact 4.3 below)

Monotone polygons can be easily triangulated [21].

Step 3(d) In $O(\log n)$ time, find tangents from p_t, the top most point of CH(p_l, p_r), to both CH(p_f, p_l) and CH(p_r, p_e).

REMARK: We are left with the polygon as shown Figure 4.2b.

Step 4 Triangulate remaining polygon.

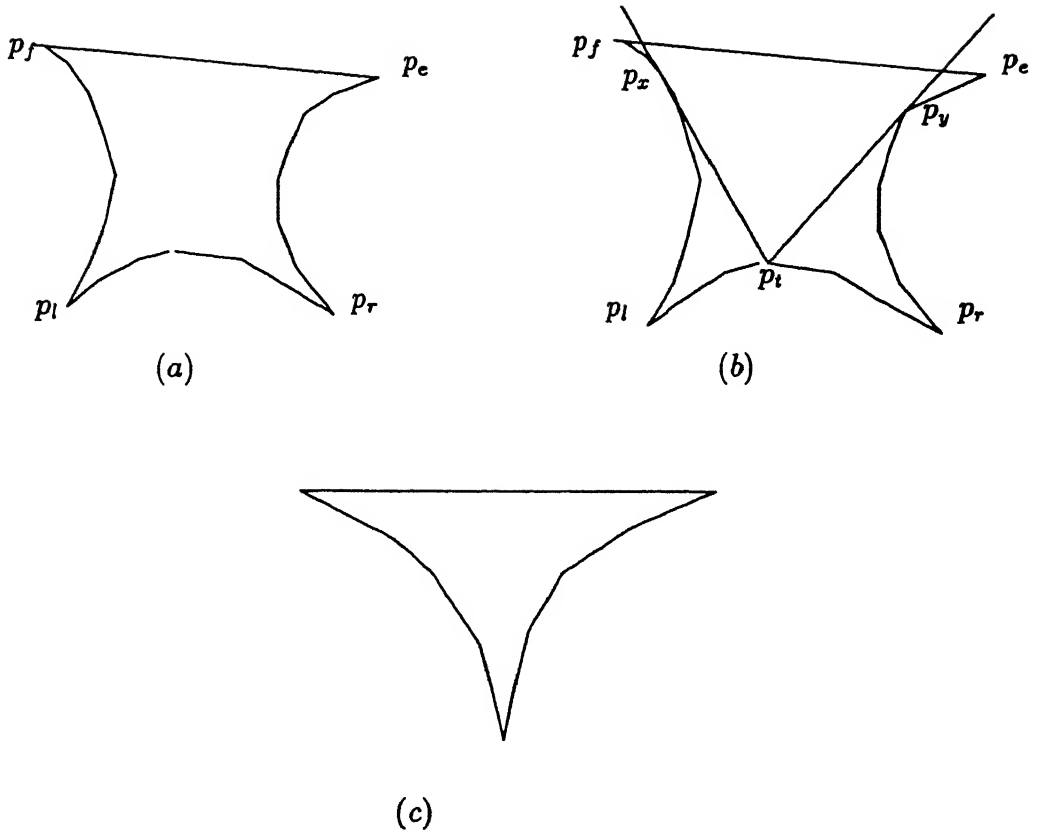


Figure 4.2: Steps during execution of algorithm, (a) The polygon P_1^i after step 3(c); (b) After step 3(d); (c) Monoconvex polygon

REMARK: We show how to triangulate remaining polygon later.

Fact 4.3 In polygon P_l^i, the chains, Chain(p_f, p_l) and Chain(p_r, p_e) are monotone with respect to y axis.

Proof: The proof is similar to proof of Fact 4.2. ∎

We next consider implementation of Step 3(b), the problem of triangulating P_{stal}, the polygon bounded by Chain(p_l, p_r) and CH(p_l, p_r). Actually P_{stal} is divided into smaller polygons by hull edges of CH(p_l, p_r). Call the part of P_{stal} that lies below any hull edge as $P_{substal}$ (See Figure 4.3a for an example of $P_{substal}$). Next we look at the problem of triangulating $P_{substal}$ (implementation of Step 3(b))

3(b).1 Identify all bottom peaks $(p_{bp1}, \cdots, p_{bpr})$ of $P_{substal}$. Find $p_{t1}, p_{t2}, \cdots, p_{tr}$, top most points (i.e stalagmitic point) between two consecutive bottom peaks. If $y(p_{ti-1}) > y(p_{ti})$ then draw a horizontal line from p_{ti} so that it touches the Chain(p_{ti-1}, p_{bpi}) at (say) p_{mpi}. Similarly if $y(p_{ti}) < y(p_{ti+1})$, then draw a line to touch the Chain(p_{bpi+1}, p_{ti+1}) at p_{mpi+1}

3(b).2 Triangulate each polygon that lies between the horizontal segment $p_{mi}p_{ti}$ and Chain(p_{mi}, p_{ti}).

3(b).3 Join two successive topstalagmitic vertices; a vertex p_{ti} is a *topstalagmitic* vertex, if $y(p_{ti-1}) < y(p_{ti}) > y(p_{ti+1})$

3(b).4 Triangulate each $P_{ssubstal}$, the sub-polygon that lies below two successive topstalagmitic vertices. The sub-polygon is a monotone polygon, (See Fact 4.4, below).

3(b).5 Triangulate remaining monotone polygon

It can be easily shown that polygon in Step 3(b).2, the polygon lying between horizontal segment $p_{mi}p_{ti}$ and Chain(p_{mi}, p_{ti}) is monotone and hence, can be easily triangulated. The remaining polygon after Step 3(b).2 is as shown in Figure 4.3b. Let p_{ti} and p_{tj} $(i < j)$ be two successive topstalagmitic vertices. To triangulate each $P_{ssubstal}$, (to implement Step 3(b).4), observe that

Fact 4.4 The chain formed by vertices $p_{ti}, p_{mi+1}, p_{ti+1}, \cdots, p_{tj}$ is combination of at most two monotone (with respect y axis) chains.

Proof: We show that in this chain (say TSChain(p_{ti}, p_{tj})) the y co-ordinate first decreases and then increases. If p_{tk} is any vertex in this chain then observe that the vertices between p_{tk} and p_{mk+1} are regular. Consider any three successive topstalagmitic vertices p_{tk-1}, p_{tk} and p_{tk+1}, then either $y(p_{tk-1}) < y(p_{tk}) < y(p_{tk+1})$ or $y(p_{tk-1}) > y(p_{tk}) > y(p_{tk+1})$.

Note that if $y(p_{tk-1}) < y(p_{tk})$ and $y(p_{tk}) > y(p_{tk+1})$, then p_{tk} becomes a topstalagmitic vertex, and this vertex marks the end of chain TSChain(p_{ti}, p_{tj}). Thus, the chain TSChain(p_{ti}, p_{tj}) can be a combination of at most two monotone chains. ∎

Thus, the polygon bounded by the line p_{ti}, p_{tj} and TSChain(p_{ti}, p_{tj}) is a monotone polygon, and can be easily triangulated; this completes the description of Step 3(b).4 The remaining polygon (after Step 3(b).4) is monotone as the chain formed by joining two successive topstalagmitics is monotone. As Step 3(b).5 can be done in optimal $O(\log n)$ time, entire Step 3(b) can be implemented in $O(\log n)$ time with $\frac{n}{\log n}$ processors.

Coming to implementation of Step 4, let the tangent from the top point (say p_t) of CH(p_l, p_r) touch hulls CH(p_f, p_l) and CH(p_r, p_e) at p_x and p_y respectively. As sub-chain Chain(p_f, p_x) of chain CH(p_f, p_l) is monotone, and also as sub-chain Chain(p_y, p_e) of chain CH(p_r, p_e) (by Fact 4.3) are monotone, and the polygon formed by Chain(p_f, p_x) the line p_xp_t, the line p_tp_y and the Chain(p_y, p_e) is also monotone

and can be easily triangulated.

We have to triangulate the polygon that is bounded by Chain(p_x, p_l) of CH$(p_f, p_l)$and Chain(p_l, p_t) of CH(p_l, p_r) and the tangent $p_t p_x$. The polygon has two convex chains (call them the left and right chains) sharing a common vertex, and a straight line joining the other ends of the chains. We call such polygons *monoconvex* polygons (see Figure 4.2). Similarly the polygon bounded by Chain(p_t, p_r) of CH$(p_t, p_r)$and Chain(p_r, p_y) of CH(p_r, p_y) and the tangent $p_t p_x$ is a monoconvex polygon. Monoconvex polygons can be triangulated as follows:

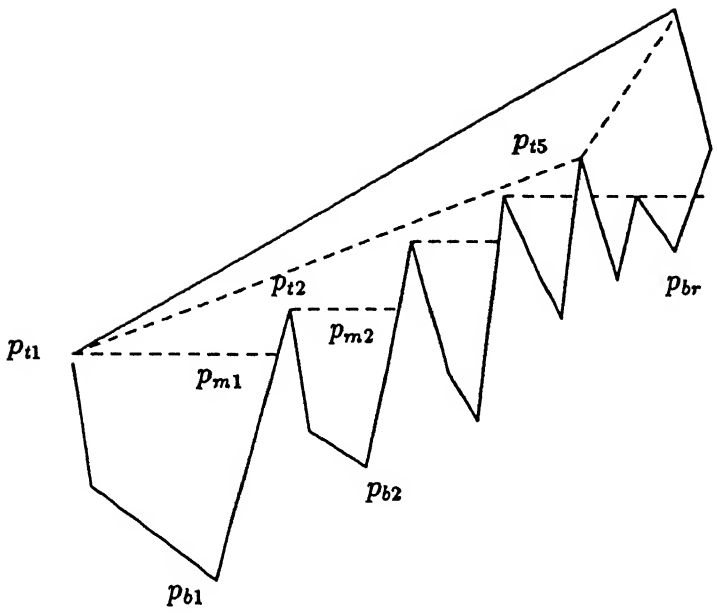
4.1 Find tangents from each point of right chain to the left chain.

4.2 Let tangents from the points p_i and p_{i+1} pass through p_j and p_k respectively. Then Join the point p_i with all the vertices that lie between p_j and p_k.

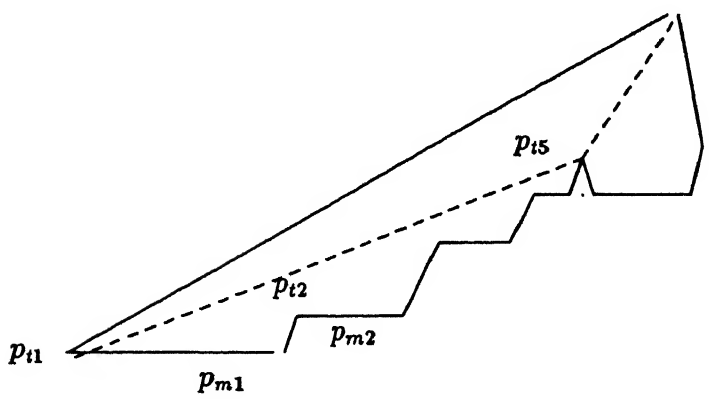
Observe that the vertices of right chain satisfy Property 2.1 (Chapter 2), as the right chain is a convex chain. This implies that Step 4.1 can be optimally implemented using the algorithm for finding tangents from a set of points to a convex hull (see Chapter 2).

Hence the following theorem can be stated:

Theorem 4.1 A star shaped polygon can be triangulated in $O(\log n)$ time with $\frac{n}{\log n}$ processors on a CREW PRAM.



(*a*)



(*b*)

Figure 4.3: (*a*) The polygon $P_{substal}$, p_{t1} and p_{t5} are consecutive topstalagmitics; (*b*) $P_{substal}$ after Step 3(b).2; Polygon $p_t1, p_m1, \cdots, p_t5, p_t1$ is $P_{ssubstal}$

Chapter 5

Convex Hull of a Simple Polygon

5.1 Introduction

Finding convex hull of a set of points is a well studied problem in Computational Geometry. Several algorithms are available to solve this problem efficiently on a sequential computational model [29]. As sorting is linear time reducible to convex hull problem convex hull problem has a lower bound of $\Omega(n \log n)$. Moreover if the points are vertices of a simple polygon then the problem becomes much simpler, and can be solved in linear time [29]. Convex hull can also be computed in linear time, in case points are sorted (with respect to one of the co-ordinates) or are vertices of a monotone chain.

On parallel computational models also, the problem of finding convex hull has been extensively studied [2]. Chow gave an algorithm that works in $O(\log^2 n)$ time with n processors. Atallah and Goodrich [5] have obtained an optimal $O(\log n)$ time algorithm on CREW model, to find the convex hull of a set of points in the plane. They use rootish divide and conquer technique to bring down the time complexity. Aggarwal *et al.* [1] also describe an optimal algorithm for the same problem. Cole and Goodrich [13] propose an optimal algorithm for the same problem, without using the square root function. They use cascading divide and conquer paradigm [4] for merging of sub-problems in $O(\log n)$ time. An optimal $O(\log n)$ time algorithm with $\frac{n}{\log n}$ processors is given by Goodrich [20] to compute the hull when the points are sorted. He brings down the processors by stopping the recursion after sometime and then solving the sub problems sequentially. The algorithm also works when the points are vertices of a monotone chain. Miller and Stout [28] obtain an optimal $O(\log n)$

time algorithm on EREW model, to compute the convex hull. The problem of finding convex hull of a simple polygon is considered by Wagner [19, 37] and he proposes an optimal O($\log n$) time algorithm for it. Here in this chapter we obtain the same bounds, but the technique used is different. We use the concepts of visibility to solve the problem. Atallah *et al.* [3] describe an optimal $O(\log n)$ time algorithm with $\frac{n}{\log n}$ processors on EREW PRAM model, for finding visibility polygon from a point. Another result needed is an optimal $O(\log n)$ time algorithm with linear cost (on CREW) by Goodrich *et al.* [22] for finding shortest path in a triangulated polygon. In this chapter we use CREW PRAM model of computation.

We assume that the (simple) polygon P is given as counter clockwise sequence of vertices (i.e., their respective x and y coordinates) p_1, p_2, $\cdots$, p_n. The symbol P denotes both the boundary of the polygon (bd(P)) as well as the region bounded by it. Two points in P are *visible* from each other if and only if the segment joining them lies completely inside P. The *visibility polygon* of P from a point c is that part of p which is visible from c. Let cp_i be a tangent from a point c to convex hull of P. The segment cp_i is called *right(left) tangent* of P if all points of P lie to the right (left) of the line passing through c and p_i. Unless stated otherwise in the chapter, line passing through p_i and p_j direction will be from p_i to p_j. Chain(p_i, p_j) denotes the boundary formed by the vertices p_i, p_{i+1}, $\cdots$, p_j in counter clockwise order. UH(P) will denote the upper hull of polygon P. The left most and right most points of P will be denoted by p_l and p_r respectively.

5.2 The algorithm

If C is convex hull (with vertices $v_1, \cdots, v_m$ in counter clockwise direction) and v is a point outside it then observe that:

Fact 5.1 If the left and right tangents to C from v passes through v_l and v_r then all the vertices in Chain(v_l, v_r) of C are visible from v.

Fact 5.2 If the left (right)tangent from v to C passes through v_l (v_r) then v lies to the right(left) of $v_{l-1}v_l$ ($v_{r-1}v_r$) and to the left (right)of v_lv_{l+1} (v_rv_{r+1})

Now we describe the algorithm to compute the convex hull of P.

1. Find p_l and p_r, the left most and right most points of P.
2. Consider all the points p_i of P. Compute the anti-clockwise angle (clockwise) angle between lines p_lp_i and p_lp_r. Let p_c (p_d)be the point for which the angle is

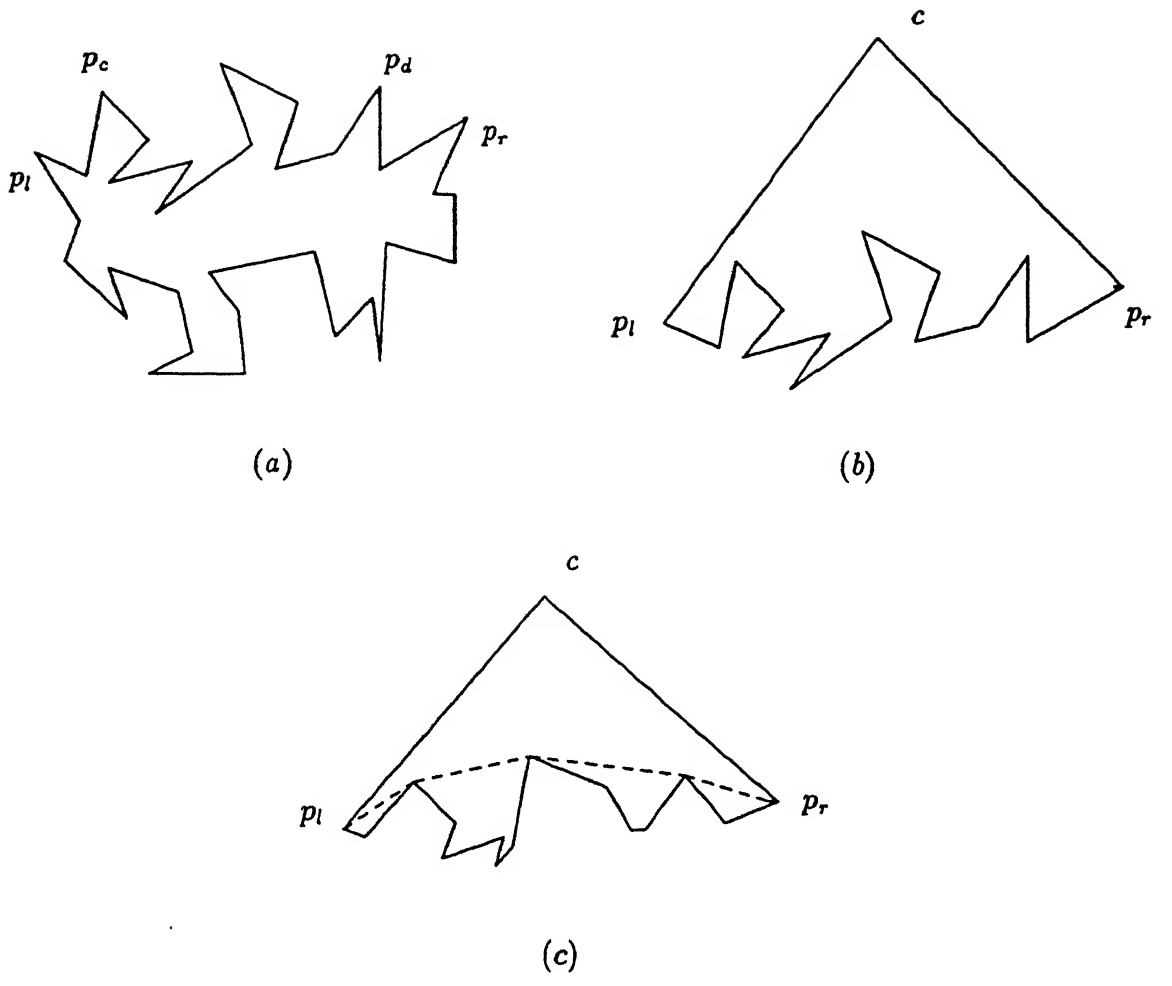


Figure 5.1: (*a*): The polygon P; (*b*): The polygon P'; (*c*): The polygon p''.

maximum.

3. Let c be a point that lies to left of $p_l p_c$ and to the right of $p_r p_d$. Join c with p_l and p_r forming a new polygon (P') $c, p_l, p_{l+1}, \cdots, p_r, c$. Find the visibility polygon $P''$of P' from c
4. Triangulate the polygon P''.
5. Find the shortest path from p_a to p_b in P'', which is the upper hull.

First observe that

Claim 5.1 If a point p_i belongs to the upper hull of P then it is visible from c.

Proof: The points p_l and p_r are present in the upper hull of P. Let C be the convex polygon formed by UH(P) and the line segment $p_r p_l$. The point c lies to left of the line passing through p_l and p_c, it also lies to the right the line passing through $p_r p_l$. Thus due to Fact 5.2 tangent to C from c passes through p_l. Similarly we can see that

the right tangent from c to C passes through p_r. Consider the polygon P_c formed by cp_l, UH(P), and p_rc. No other point of P can properly lie (i.e. it should not even touch the edges) inside P_c, because if it lies then it should belong to UH(P). Now there are no objects (polygon edges) in side P_c. Tangents from c pass to C through p_l and p_r. By Fact 5.1 all points in the chain Chain(p_l, p_r) of C are visible from c. Thus all the points in UH(P) are visible from c. ∎

Theorem 5.1 Convex hull of P can be computed in $O(\log n)$ time with $\frac{n}{\log n}$ processors.

Proof: First we prove the correctness of the algorithm. By claim5.1 upper hull of P is same as upper hull of P'' (formed during step 3). So we will show that for the polygon P'' (formed in step 3) convex hull and shortest path from p_l to p_r are same. Observe that if p_ip_j is an edge of the hull UH(P) then p_i is visible to p_j in the polygon P'. So if p_ip_j is an edge of UH(P) then shortest path from p_i to p_j in P'' is nothing but the edge p_ip_j. We prove that if p_ip_j and p_jp_k are two consecutive edges of UH(P) then shortest path from p_i to p_k in P'' is $p_i - p_j - p_k$. First we prove that p_k is not visible from p_i in P_i''. Draw a straight line from p_i to p_k, note that p_k should lie above this line, as p_i ,p_j and p_k are consecutive edges of convex hull. Now there is a chain, Chain(p_i, p_j) (of P'') below the hull edge p_ip_j, similarly there is a chain Chain(p_j, p_k) (of P_i'') below the hull edge p_jp_k. So the horizontal line segment from p_i to p_k intersects both these chains, so p_i is not visible from p_k. This implies that the shortest path from p_i to p_k can not lies below p_ip_j nor can it lie below p_jp_k. So it should lie above (or on) these edges. This means the shortest path from p_i to p_k should lie in P''. But in P'' there are no vertices of P other than the vertices that are in UH(P). So the shortest path from p_i to p_k should pass through the vertices of UH(P) only. As UH(P) is convex chain the shortest path from p_i to p_k should pass through p_j. So the shortest path between p_i and p_k is p_i - p_j - p_k, which is equal to convex hull of Chain(P_i, P_j) (of P''). With similar arguments we can show that the shortest between p_l and p_r is equal to convex hull of Chain(p_l, p_r) of P''. This proves the correctness of the algorithm. We shall now see the implementation details.

Steps 1 and 2 can be done trivially in $O(\log n)$ time with $\frac{n}{\log n}$ processors. Step 3 can be done using the algorithm of [3] with $\frac{n}{\log n}$ processors in $O(\log n)$ time. In step 4 we have to triangulate the polygon P''. This is a star shaped polygon as we have found the visibility polygon of P' from c to form P'' and the star point is c. So we can triangulate it easily, we have to just join all the vertices of P'' with c. We can

find the shortest path in a triangulated polygon in $O(\log n)$ time optimally [22] So the entire algorithm works in $O(\log n)$ time with $\frac{n}{\log n}$ processors. ∎

Chapter 6

Visibility Polygon from a Point

6.1 Introduction

Finding visible regions of a polygon from a source is well studied problem in computational geometry. Several linear algorithms have been obtained for finding visible regions of a simple polygon from a point source or from a line source. Lee [25] shows that visibility polygon from a point can be computed in linear sequential time. Avis and Toussaint [6] described a linear time algorithm to test whether a given polygon is weakly visible from a given edge of the polygon. Suri and Sack [31] obtain a linear time algorithm to test whether a given polygon is weakly visible from any edge of the polygon. Guibas *et al.* [23] proposed linear time algorithm for finding weak visibility polygon from a segment inside polygon. Ghosh [16] extends the notion of visibility from a point or line segment to visibility from convex set, to get linear time algorithms for computing complete, strong and weak visible polygons from a convex set.

Attempts have also been made to obtain parallel algorithms for these problems. Atallah *et al.* [3] obtain an optimal $O(\log n)$ time algorithm for finding visibility polygon from a point on the Exclusive Read Exclusive Write (EREW) model. Chandru *et al.* [10] propose an $O(\log(m+n))$ time algorithm with $(m+n)$ processors on the CREW model for finding complete visibility polygon.

Here we give some fast parallel algorithms for computing visibility polygon of a simple polygon from a point. We first describe an $O(\alpha(n))$ time algorithm with n^2 processors. Then we propose a fast algorithm for computing the visibility polygon in $O(\log\log n)$ time. This algorithm uses $\frac{n^2}{\log n}$ processors. We propose another algorithm which takes $O((\log\log n)^2)$ time with $\frac{n^{1+\frac{1}{2}}}{\log n}$ processors. We also describe an optimal

$O(\log n)$ time algorithm.

We assume that the (simple) polygon P is given as (array of) counter clockwise sequence of vertices (i.e., their respective x and y coordinates) $p_1, p_2, \cdots, p_n$. The symbol P denotes both the boundary of the polygon ($\mathrm{bd}(P)$) as well as the region bounded by it. Two points in P are *visible* from each if and only if the segment joining them lies completely inside P. If c is a point in P then the *visibility polygon* of P from c is the set of all points in P that are visible from c. $\mathrm{Chain}(p_i, p_j)$ denotes the boundary formed by the vertices $p_i, p_{i+1}, \cdots, p_j$ in anti-clockwise order. $\mathrm{UChain}(p_i, p_j)$ denotes the boundary formed by the vertices between p_i and p_j; the direction can be either clockwise or anti-clockwise. In this chapter, the point c will be assumed to lie inside the polygon P.

In Section 6.2 we describe the $O(\alpha(n))$ time algorithm. In the Section 6.3 we propose an $O(\log\log n)$ time algorithm that uses $\frac{n^2}{\log n}$ processors. In Section 6.4 we describe an algorithm which uses $\frac{n^{1+\frac{1}{c}}}{\log n}$ processors and takes a time of $O((\log\log n)^2)$. In the same section we show that this algorithm can be easily modified to give an optimal $O(\log n)$ time algorithm.

6.2 $O(\alpha(n))$ time algorithm

First we consider the following "trivial" algorithm. The algorithm takes $O(\alpha(n))$ time with n^2 processors on a CRCW PRAM.

Let c be the point from which we are computing the visibility polygon.

0. Initialise the visibility polygon P' to be P.
1. For each vertex p_i of the polygon P, check whether the segment cp_i intersects any edge of the polygon, if cp_i intersects an edge then mark p_i as 'invisible' else 'visible'.
2. For each vertex p_i find the (first) vertex p_j $(j > i)$ such that p_j is visible from c, and no other vertex in $\mathrm{Chain}(p_i, p_j)$ is visible from c.
3. Find the intersection of the edge $p_{j-1}p_j$ with the segment cp_i. Let p'_{j-1} be the intersection point; in P', replace p_{j-1} with p'_{j_1} and mark p'_{j-1} 'visible'.
4. The polygon formed by joining adjacent 'visible' vertices of P' gives the visibility polygon of P from c.

Now we will show that the algorithm correctly computes the visibility polygon

from c. In first step we have identified the vertices of P that are visible from c. Consider any vertex p_j that is marked 'visible', let p_j be the first 'visible' vertex that we encounter in counter clockwise walk from p_i. Now we have to compute the visible part of Chain(p_i, p_j). We know that no polygon vertex that is in Chain(p_i, p_j) is visible from c. Because if any vertex in that chain is visible from c then p_j cannot be the first visible vertex encountered during the anti-clockwise walk from p_i. So the visible part of Chain(p_i, p_j) is just the visible part of edge $p_{j-1}p_j$. Hence, it is sufficient to compute only the intersection of edge $p_{j-1}p_j$ with cp_j. Thus, the polygon formed by visible vertices gives the visibility polygon of P from c.

To analyse the algorithm, assume that we have n^2 processors at our disposal. Assign n processors to each vertex of the polygon. We can test whether a point p_i is visible from c in constant time using n processors assigned to p_i; basically, we have to test whether the segment cp_i intersects any side of the polygon. As there are n edges in polygon, we can test in constant time. Step 2 can clearly be done in constant time. All vertices marked 'visible' in the output array, give the desired polygon. To delete vertices that are not visible from c, for each visible vertex we have to find the nearest left and nearest right vertices that are marked 'visible'. This is equivalent to the problem of finding nearest zeros in a binary array and can be solved in $O(\alpha(n))$ time either using the recursive star data structure of Berkman and Vishkin [9] or using the algorithm of Ragde [30]. Hence, we can find the visibility polygon of P from any point c in $O(\alpha(n))$ time using n^2 processors.

6.3 The $O(\log \log n)$ time algorithm

Assume that we have divided the polygon into z sub-chains $P_1, \cdots, P_z$. Let P_i' be the visibility chain (part of chain visible from c) of P_i from c; in definition of P_i', we do not consider other chains P_j, $j \neq i$ (i.e., each P_j is not looked upon as an obstacle, but is assumed to be transparent). Let p_l be the first and p_r be the last point (end point) of the chain, P_i' respectively.

Let P_i'' be the polygon formed by adding segments (c, p_l) and (c, p_r) to P_i'. Consider the intersection of polygon P_i'' with chains P_j''s ($1 \leq j \leq z$, and $j \neq i$). Consider the parts of these chains that lies inside P_i'' (they can touch the edges of P_i''); we call these parts Sub$p_i$chains (here P_i indicates that these chains are inside P_i'', similarly we can have Sub$p_j$chains). We will divide these Sub$p_i$chains into three types based

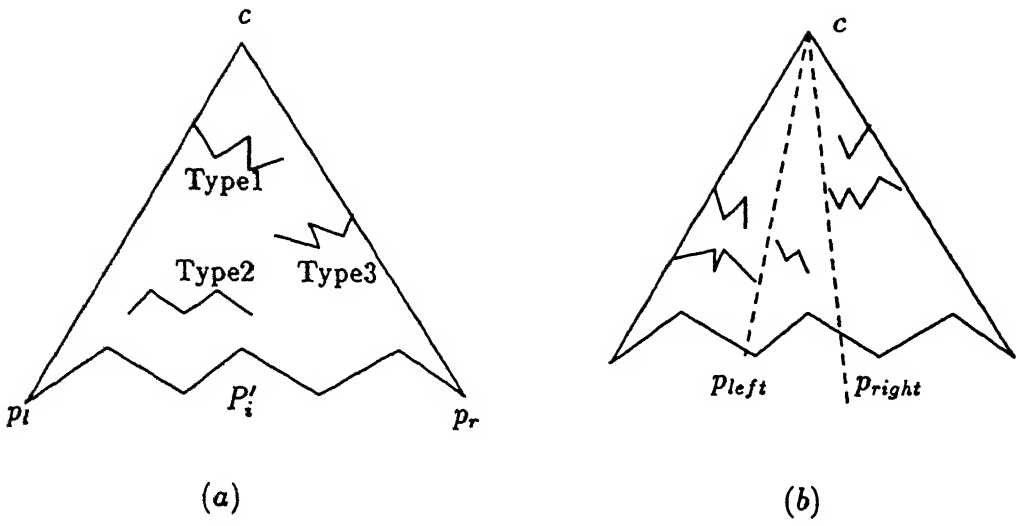


Figure 6.1: *a* : Types of Sub$p_i$chains. *b* : Forming P_i'''

on their intersection type with P_i'' (see Figure 6.1a).

Type 1: The Sub$p_i$chain touches segment cp_l.

Type 2: The Sub$p_i$chain lies inside polygon P_i''. It touches neither cp_r nor cp_l.

Type 3: The Sub$p_i$chain touches cp_r.

Let p_j (p_i) be any vertex that belongs to a Type 3 (Type 1) chain; and the line cp_j (cp_i)makes an angle (anti-clockwise direction) of Ang_j (Ang_i) with cp_l. Find the vertex $p_{\min}$ ($p_{\max}$) such that the value of $Ang_{\min}$ ($Ang_{\max}$) is minimum (maximum) among all Ang_j's (Ang_i's).

Let the line $cp_{\max}$ ($cp_{\min}$) intersects P_i' at p_{left} (p_{right}). Note that the vertices in Chain($p_l p_{\max}$) and also the vertices in Chain($p_{\min} p_r$) are not visible from c (Figure 6.1b). Let the polygon formed by $cp_{\max}$, Chain($p_{\max} p_{\min}$) and $p_{\min}c$ be P_i'''.

Consider any Sub$p_i$chain that lies inside polygon P_i'''. Let x (y) be the point of Sub$p_i$chain such that cx (cy) makes minimum (maximum) anti-clockwise angle with cp_l among all other vertices belonging to the same chain. Call this minimum (maximum) angle as StartAngle (EndAngle). If the StartAngle of one Sub$p_i$chain is equal to the EndAngle of another Sub$p_i$chain then treat them as a single chain, and call it ContinuousSub$p_i$chain.

Figure 6.2a gives an example of ContinuousSub$p_i$chain. In the figure there are two Sub$p_i$chains Chain(p, q) and Chain(r, s). Here StartAngle of one chain is equal to EndAngle of another chain. So we should treat both of them as a single ContinuousSub$p_i$chain.

Lemma 6.1 There can be at most two ContinuousSub$p_i$chains in P_i'''.

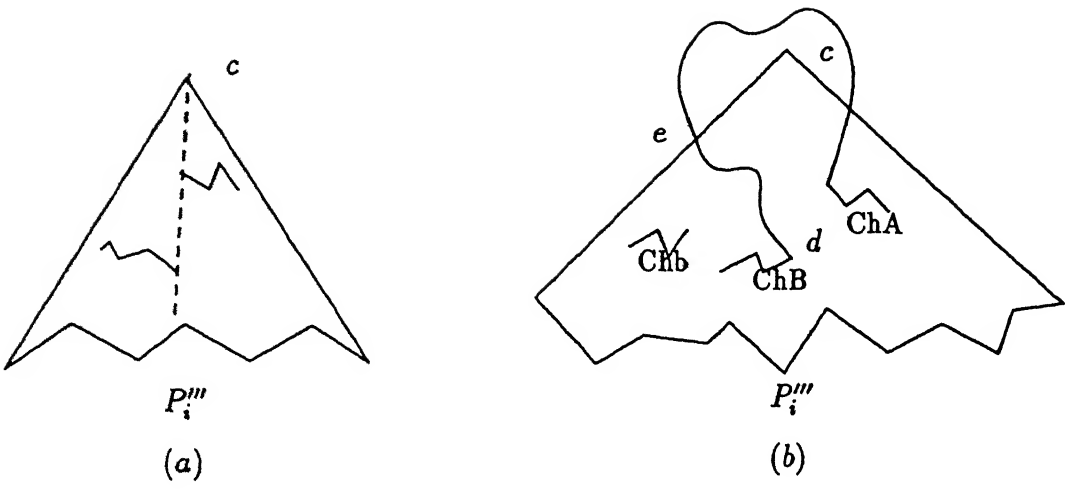


Figure 6.2: a : Example of ContinuousSub$p_i$chains. b : Proof of Lemma 6.1

Proof: We will prove the Lemma by contradiction. Assume that there are three ContinuousSub$p_i$chains (call them ChA, ChB and ChC) in P_i''' (Figure 6.2b). There should be a polygon chain from one end point of ChA to ChB, but it cannot completely lie inside P_i'''; as in that case ChA and ChB will be in the same ContinuousSub$p_i$chain. So let us assume that the chain from ChA to ChB does not completely lie in P_i''' (Figure 6.2b), In this case we can merge ChB, ChC and Chain(c, d) to form a ContinuousSub$p_i$chain. Hence, in both cases we cannot have three ContinuousSub$p_i$chains. A similar proof can be given for non existence of more than three ContinuousSub$p_i$chains. Hence, the Lemma follows. ∎

By Lemma 6.1, to find the visible portion of P_i (or P_i') in the polygon P it is sufficient to find the end points of the two ContinuousSub$p_i$chains (in case there are two chains, otherwise one). Thus, we can find the region not visible from c (these chains block some part of P_i) and hence the region of P_i visible from c. Less informally. the algorithm is:

1. Divide the polygon chain into $\frac{n}{\log n}$ parts each having at most $\log n$ points and assign n processors to each chain P_i.
2. For each chain P_i find the visibility chain P_i' of P_i from c. While computing visibility chain of P_i ignore all other chains (i.e. consider edges of other chains as transparent).

   REMARK: As each chain P_i has at most $\log n$ points, P_i', the visibility chain of p_i can be computed in $O(\alpha(n))$ time using the algorithm, of previous section.
3. For each chain P_i', join both its end points to c to form polygon P_i''. Find the

intersection of P_i'' with all other chains.

REMARK: Now we have Sub$p_i$chains for each P_i'

4. Find polygon P_i''' (as defined in proof of Lemma 6.1) for each i.
   REMARK: Sub$p_i$chains can be either Type 1, Type 2 or Type 3. In case of Type 1 and Type 3, Sub$p_i$chains, we can form polygon P_i''' by finding vertices (say x and y) that make minimum (among Type 3 vertices) and maximum (among Type 1 vertices) angles with line cp_l. Find intersection of cx and cy with P_i' to get vertices $p_{\max}$ and $p_{\min}$. We can find the intersection of these lines with P_i' (in constant time) as there are at most $\log n$ points in P_i' and we have n processors. This gives the desired polygon P_i'''. Type 2 Sub$p_i$chains will be inside the polygon P_i'''.
5. Identify the ContinuousSub$p_i$chains and find visible regions of P_i in P.
6. Club the remaining regions of P_i' to form the visibility polygon of P from c.

Claim 6.1 Above algorithm can be implemented in $O(\log\log n)$ time with $\frac{n^2}{\log n}$ processors.

Proof: Step 1 can clearly be done in constant time with n processors. In Step 3, we have to find intersection of P_i'' with all other chains. There are $\frac{n}{\log n}$ chains, and each P_i'' has n processors, thus we assign $\log n$ processors to each pair (p_i'', p_j').

We find intersection between each pair with $\log n$ processors as follows. Recall, that all chains are star shaped (as each chain is visible from c), and each chain has at most $\log n$ points. So the problems reduces to that of finding the intersection between two star shaped polygons each having at most $\log n$ points with $\log n$ processors. Maheswari and Ghosh [17] have shown that the problem of finding intersection of two star shaped polygons each having at most m points can be solved, in $O(\log m)$ time with $\frac{m}{\log m}$ processors. Thus, the problem of finding the intersection between each pair takes $O(\log\log n)$ time. As we have to find intersection between P_i'' with all other chains, all intersections can be found in $O(\log\log n)$ time with n processors. Thus, for all P_i''s Step 3 can be done with $\frac{n^2}{\log n}$ processors in $O(\log\log n)$ time.

In Step 4, the functions minimum and maximum can be computed in $O(\log\log n)$ time on CRCW model.

In Step 5 we have to find the visible region of P_i' in P (considering the existence of all other chains; that is we assume that they are "opaque"). We will do this without explicitly computing the ContinuousSub$p_i$chains. In P_i''' there can be at most $\frac{n}{\log n}$ Sub$p_i$chains. We have $\log n$ processors assigned to each Sub$p_i$chain. Let Ch_i be any

Sub$p_i$chain with end points r_i and s_i (cr_i makes smaller angle than cs_i with the line cp_l). Find the intersections u_i and v_i of cr_i and cs_i with P_i'. From now on we treat u_i and v_i also as vertices of P_i'. Mark all the vertices (including u_i and v_i) and edges of P_i' as "visible". Assume that with each vertex we can store two more data items, left item and right item. Now make Ch_i write *right* in "slot" for right item of u_i and *left* in the "slot" for left item of v_i. All concurrent writes are of "COMMON" type. Let $ChA($ with end points ChA_f and ChA_e) and ChB (with end points ChB_f and ChB_e) be two ContinuousSub$p_i$chains.

Assume that $cChA_e$ ($cChB_f$) makes larger (smaller) angle than $cChA_f$ ($cChB_e$) with the line cp_l. Let the intersection of $cChA_e$ ($cChB_f$) with P_i' be Pa_i (Pb_i). Note that the right item of Pa_i will be empty, while the left item is filled with *left*. Also note that by Lemma 6.1 no other vertex can have this property. Similarly left item of Pb_i will be empty, while right item is filled with *right*. And again by Lemma 6.1, this also cannot happen for any other vertex. So the visible portion of P_i' in P is the chain between Pa_i and pb_i.

To implement Step 6, we delete the invisible parts of chain P_i'. The successor of Pa_i will be ChA_e and the predecessor of Pb_i will be ChB_f. To delete invisible parts efficiently we maintain chain P_i' as a binary search tree. Chain P_i' can be maintained as a binary search tree because vertices of P_i' are angularly sorted about c. Again by Lemma 6.1, we have to delete at most two sub-chains of P_i'. These trees are exactly identical to the hull trees maintained by Goodrich [20] and deletions and merging of trees can be carried as in [20]. ∎

Thus, the following theorem follows:

Theorem 6.1 The visibility polygon of P from a point c lying inside P can be found in $O(\log\log n)$ time, with $\frac{n^2}{\log n}$ processors on CRCW PRAM.

6.4 Reducing the number of processors.

In this section we describe two algorithms, the first algorithm takes $O((\log\log n)^2)$ time with $\frac{n^{1+\frac{1}{\epsilon}}}{\log n}$ processors. Second is an optimal $O(\log n)$ time algorithm.

6.4.1 The algorithm with $\frac{n^{1+\frac{1}{\epsilon}}}{\log n}$ processors

In this section we reduce number of processors to $\frac{n^{1+\frac{1}{\epsilon}}}{\log n}$ (ϵ is a constant ≥ 1) with slight increase in time to $O((\log\log n)^2)$. Observe that in the previous algorithm

the costliest step is finding the intersection of two star shaped polygons. Here we will exploit a new property of these star shaped polygons to find their intersection efficiently; we use the fact that these star shaped chains are sub-chains of the original polygon P.

Let us draw a horizontal line through c. Let p_l^i (and p_r^i) be the point of P_i' such that the line cp_l^i (respectively line cp_r^i) makes the minimum (maximum) anticlock wise angle with the horizontal through c, among all other points of p_i'. Similarly, we can define p_l^j and p_r^j for chain P_j.

Lemma 6.2 Let P_i and P_j be two chains of polygon P. Let P_i' and P_j' be visibility chains of P_i and P_j respectively. Then chain P_j' can intersect chain P_i' at most twice.

Proof: We prove the Lemma by contradiction. Assume that P_j' intersects P_i' at three points (Figure 6.3a). Let the edges of P_j' that intersect P_i' be p_ap_b, p_cp_d and p_ep_f. As P is a simple polygon, none of these edge can be the edges of P. These edges were added when we computed the visibility chain of P_j' from c. This implies that there exist a polygon chain (say) UChain(p_a, p_b) from p_a to p_b; similarly there exist two other chains UChain(p_c, p_d) and UChain(p_e, p_f).

In first case, the UChain(p_c, p_d) is as shown (Figure 6.3a). In this case, polygon chain P_j should start at p_c; otherwise the chain, Chain(p_g, p_c) will be a part of P_j and will obstructs UChain(p_f, p_l^j); as a result, the edge p_ep_f will not be there. So we can assume that P_j starts at p_c.

Traverse polygon P in anti-clockwise direction, starting from p_c. We will first encounter p_e (and not p_f, p_d, p_b, or p_a). Now there should exist a chain joining p_e and p_f (UChain(p_e, p_f)), and the chain should intersect either cp_l or cp_r (Figure 6.3a); observe that this chain should be a sub-chain of P_j. In either case P_j' cannot be visibility chain of P_j from c; because if UChain(p_e, p_f) intersects cp_l then UChain(p_e^j, p_f) is not visible; and if it intersects cp_r then the UChain(p_d, p_b) is not visible. So either edge p_ep_f or edge p_ap_b of P_j' cannot exist.

Similarly, in the other case, when the UChain(p_c, p_d) is as shown in Figure 6.3b, we can prove that the edge p_ap_b cannot exist. So P_j' cannot intersect P_i' three times. Similar arguments hold when the number of intersections are more than three. So P_j' and P_i' can intersect at most at two points. ∎

In subsequent discussion we assume with out loss of generality that index p_r^i is greater then p_l^i (otherwise interchange the role of "minimum index" and "maximum index" in following discussion). Let the angles made by the lines cp_l^i, cp_r^i, cp_l^j and cp_r^j

with the horizontal line be α, β, γ, and θ. The intersection between P_i' and P_j' can be any of the following six cases based on the relations between these angles.

Case 1 ($\gamma < \alpha < \beta < \theta$): There are three sub-cases as the chains may or may not intersect (see Figure 6.4c).

Case 2 ($\gamma < \alpha < \theta < \beta$): There are two sub-cases, as the chains may or may not intersect (see Figure 6.4a).

Case 3 ($\gamma < \theta < \alpha < \beta$): In this Case the chains do not intersect (see Figure 6.4e)

Case 4 ($\alpha < \gamma < \beta < \theta$): Here there can be two sub-cases, as the chains may or may not intersect. This is symmetric to Case 2 (see Figure 6.4b).

Case 5 ($\alpha < \gamma < \theta < \beta$): Here also there are two sub-cases, as the chains may or may not intersect (see Figure 6.4d)

Case 6 ($\alpha < \beta < \gamma < \theta$): Here the chains do not intersect (see Figure 6.4e).

Observe that there cannot be any other case (or sub-case). For each case we show how to find the intersection between P_i' and P_j'.

Cases 3,6 : Here there is no intersection.

Case 5 : Here we have to find the intersection point of the line cp_l^j with the chain P_i'. If the intersection does not lie on the segment cp_l^j then P_i' and P_j' do not intersect, otherwise P_j' lies in P_i''.

Case 1 : In this case P_i' is not visible from c or the intersection is nil; The case when the intersection is nil can be easily detected with binary search. If the intersection is not nil then we prove that P_i is not visible from c as follows:
If P_j' is as in Figure 6.6a then P_i is obviously not visible from c. However, if P_j' is as in Figure 6.6b, then P_j' intersects P_i' at two points. Let the edges of P_j' that intersect p_i' be $p_{in1}p_{in2}$ and $p_{in3}p_{in4}$; there should be a path from p_r^i to P_j'. Let the path touch P_j' at point (say) p_{start}^j. Let p_{start}^j lie in between vertices p_r^j and p_{in1} of P_j'. Now the original polygon chain of P_j should leave chain UChain(p_r^j, p_{in1}) of P_j' at some point and it should enter either UChain(p_{in2}, p_{in3}) or UChain(p_{in4}, p_l^j) (both chains are sub-chains of P_j'). Let us assume that the original polygon chain of P_j' first enter UChain(p_{in2}, p_{in3}) of P_j' (see Figure 6.6b), and then leave the chain and then enter UChain(p_{in4}, p_l^j) of P_j'. Finally, the chain should reach p_l^i to complete the polygon. The chain will reach p_l^i as shown in Figure 6.6b; thus

it can be easily observed that P'_i is not visible from c.

Note that if chain is as shown in Figure 6.6c, then c will lie outside P.

Similarly it can be shown that P'_i is not visible from c in all other remaining cases.

Cases 2,4 : As Case 2 and Case 4 are symmetric, we consider only Case 4. First find the intersection point of line cp^i_r with P'_j, if the intersection point doesnot lie on the segment cp^i_l then the two chains do not intersect. Intersection of line cp^i_r with P'_j, can be found by binary search; if we have $\frac{car(P_j)}{\log n}$ processors then binary search takes $O(\log\log n)$ time ($car(P_j)$ is number points of P_j). If line cp^i_r does not intersect P'_j, then P'_i and P'_j do not intersect. Otherwise, we have to find the edge $p_{in1}p_{in2}$ of P'_j that intersects P'_i. It is sufficient to identify either point p_{in1} or point p_{in2}. This is done using Claim 6.2 (described below).

Claim 6.2 In P'_j, the visibility chain of P_j, if neither p_{in1} nor p_{in2} have the minimum index amongst the vertices of P'_j then, the intersection between P'_i and P'_j is not required in computation of visible part of P'_i.

Proof: Let us assume that neither p_{in1} nor p_{in2} have the minimum index. There should be a polygon path from p^i_r to the vertex with minimum index in P'_j (say $p_{\min}$). Note that $p_{\min}$ cannot lie between p^j_l and p_{in2}; as in that case P'_j will not be the visibility chain of P_j (see Figure 6.5b). In case $p_{\min}$ lies in the polygon chain between p^j_r and p_{in2} (see Figure 6.5a), it can be easily seen that there exists another chain CH (say) such that the region blocked by chain CH includes the region (i.e., is a super set of the region) blocked by P'_j. Thus, we need not compute the intersection of P'_i and P'_j. ∎

This completes the description of Case 4. Case 2 is similar.

Algorithm to find visibility polygon of P from c is:

1. Divide the polygon chain into $n^{\frac{1}{\epsilon}}$ parts, each part having at most $n^{1-\frac{1}{\epsilon}}$ points. Assign $\frac{n}{\log n}$ processors to each part P_i
2. If the number points in P_i is greater than $\log\log n$ then recursively find P'_i, the visibility chain of P_i. Otherwise solve the problem sequentially.
3. To merge visibility chains $P'_1, P'_2, \cdots P'_{n^{1/\epsilon}}$, find the visible portion of each P'_i in P as follows. As there are $\frac{n}{\log n}$ processors assigned to each P'_i, we distribute these processors as follows: $\frac{n^{1-\frac{1}{\epsilon}}}{\log n}$ processors are assigned to each pair (P'_i, P'_j). Form the polygon P''_i and find the intersection of P''_i with all the P'_j's as described above.

4. Find the visible region of P_i' in P
5. Merge all remaining chains

Claim 6.3 Above algorithm can be implemented in $O((\log\log n)^2)$ time with $\frac{n^{1+\frac{1}{c}}}{\log n}$ processors.

Proof: Step 1 can trivially be done in required resource bounds. Intersection between P_i'' and P_j', in Step 3 is found by finding angles α, β, γ and θ, and identifying the intersection types as explained above. As already seen the intersection can be found using binary search in $O(\log\log n)$ time with $\frac{\max(car(P_i'),car(P_j'))}{\log n}$ processors.

Each P_i' and P_j' has at most $n^{1-\frac{1}{c}}$ points and there are $\frac{n^{1-\frac{1}{c}}}{\log n}$ processors assigned to the pair (P_i', P_j'). Thus, Step 3 can be implemented in $O(\log\log n)$ time.

Step 4 is identical to Step 5 of algorithm described in Section 3, and Step 5 of this algorithm is again identical to Step 6 of the algorithm described in Section 3. As already seen, both steps can be implemented in $O(\log\log n)$ time.

Thus the recurrence equation is

$T(n) = T(n^{1-\frac{1}{c}}) + \log\log n$

$T(\log\log n) = \log\log n$

Hence, $T(n) = O((\log\log n)^2)$. ∎

Putting everything together, we have the following theorem:

Theorem 6.2 Visibility polygon of P from c can be computed in $O((\log\log n)^2)$ time with $\frac{n^{1+\frac{1}{c}}}{\log n}$ processors.

6.4.2 Optimal $O(\log n)$ time algorithm

We next modify $O((\log\log n)^2)$ time algorithm to get an optimal $O(\log n)$ time algorithm with $\frac{n}{\log n}$ processors.

1. Divide the polygon chain into $\sqrt{(\frac{n}{\log n})}$ parts , each part having at most $\sqrt{(n\log n)}$ points. Assign $\sqrt{(\frac{n}{\log n})}$ processors to each P_i.
2. If the number points in P_i is greater than $\log n$ then recursively find P_i', the visibility chain of P_i. Otherwise solve the problem sequentially.
3. To merge visibility chains $P_1', P_2', \cdots P'_{\sqrt{n/\log n}}$, find the visible portion of each P_i' in P as follows. As there are $\sqrt{(\frac{n}{\log n})}$ processors assigned to each P_i', we distribute these processors as follows: one processor is assigned to each pair (P_i', P_j'). Form the polygon P_i'' and find the intersection of P_i'' with all the P_j''s as described above.
4. Find the visible region of P_i' in P

5. Merge all remaining chains

Claim 6.4 Above algorithm can be implemented in $O(\log n)$ time with $\frac{n}{\log n}$ processors.

Proof: All the steps in this algorithm are identical to those in the $O((\log\log n)^2)$ time algorithm, except here Step 3 (binary search) which takes $O(\log\sqrt{n\log n})$ time as we have (only) one processor for each pair.

It can be easily observed that remaining steps can also be executed in $O(\log\sqrt{n\log n})$ time. So the recurrence equation is

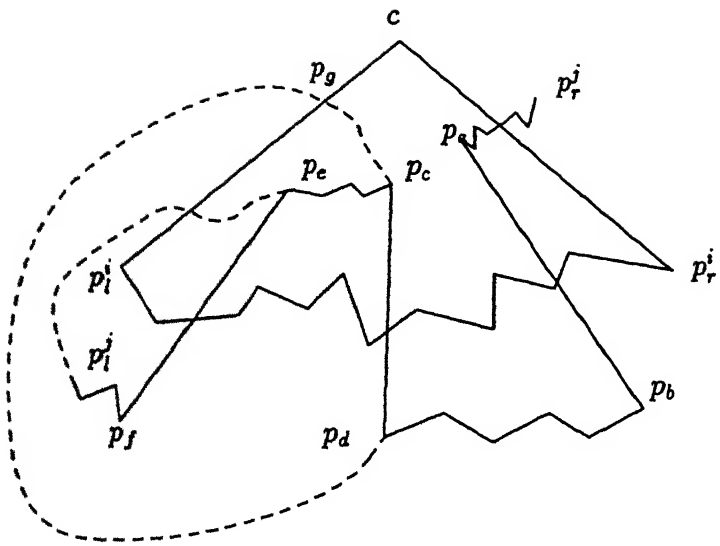
$T(n) = T(\sqrt{n\log n}) + \log(\sqrt{n\log n})$

$T(\log n) = \log n$

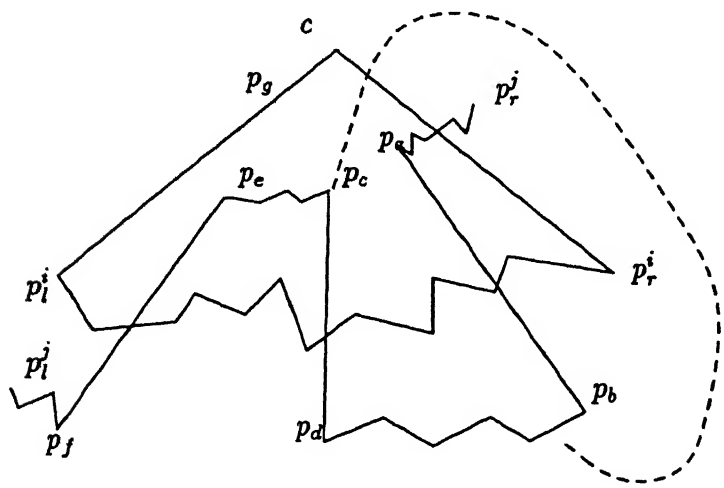
Or, $T(n) = O(\log n)$. ∎

Thus, the following theorem

Theorem 6.3 Visibility polygon of P from c can be computed in $O(\log n)$ time with $\frac{n}{\log n}$ processors.



(a)



(b)

Figure 6.3: Proof of Lemma 6.2

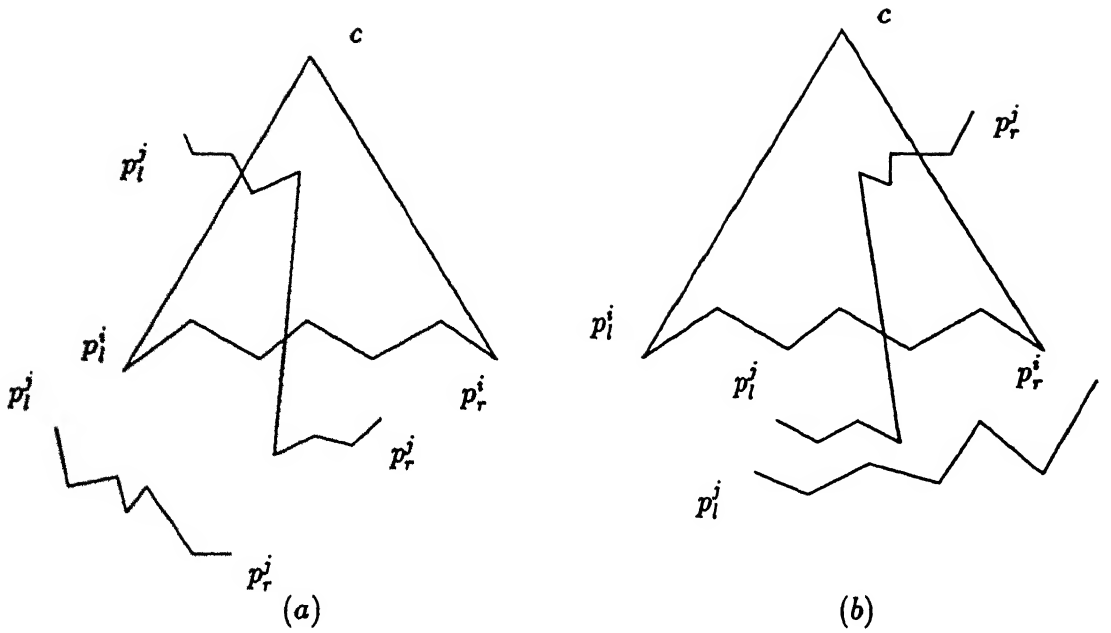


Figure 6.4: Examples of cases

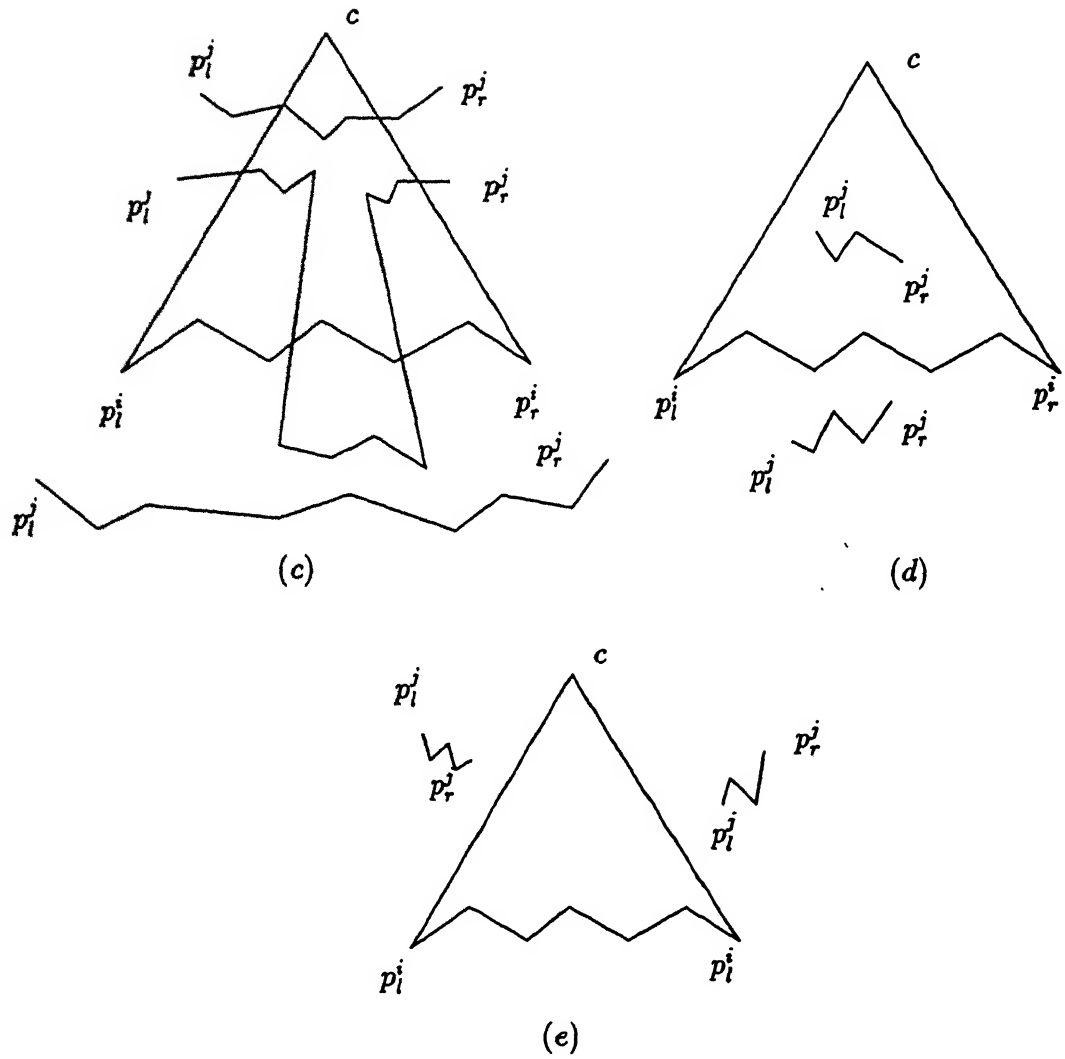


Figure 6.4: Examples of cases

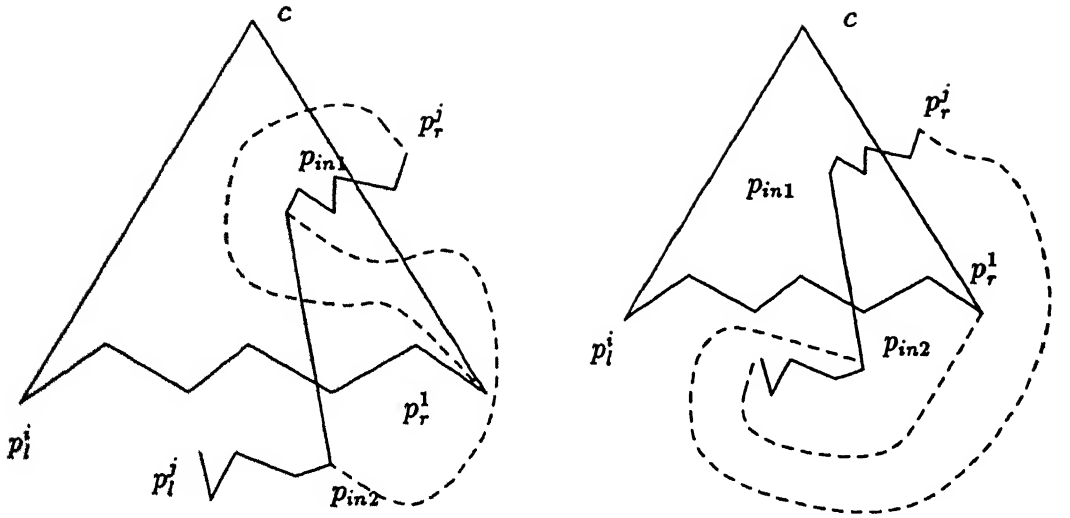


Figure 6.5: Proof of Claim 6.2

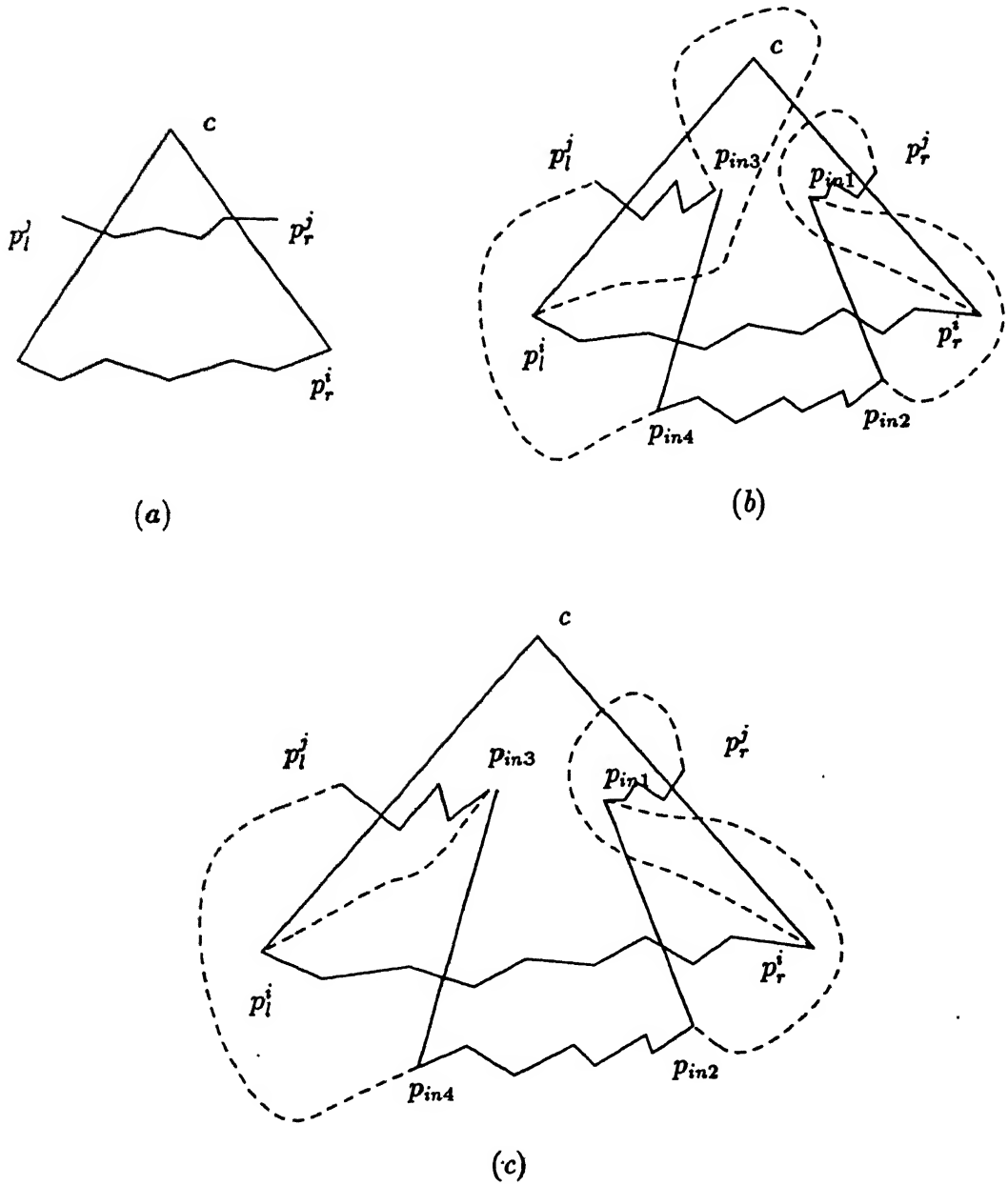


Figure 6.6: Case 1

Chapter 7

Getting Co-ordinates of Points from a Voronoi Diagram

7.1 Introduction

Suppose that we are given a Voronoi Diagram G of a point set S. In this chapter, we consider the "reverse" problem of getting the co-ordinates of the point set S from the Voronoi Diagram. This problem may arise when the point set S is lost, but the Voronoi Diagram remains, and we have to regenerate the point set.

Voronoi diagram of a point set S, is the partition of the plane into regions such that each region is the locus of points (x,y) closer to a point of S than any other point of S. Each point of the set S is associated with a region (say), Voronoi region of the diagram. To be more formal *Voronoi region* associated with point p_i (say $V(p_i)$) is intersections of half planes $H(p_i,p_j)$ $(1 < j < n$ and $j \neq i)$, where $H(p_i,p_j)$ is the half plane containing p_i that is defined by the perpendicular bisector of line p_ip_j. Voronoi diagrams are used to solve proximity problems like nearest-neighbor search [29]. Given a point set S its Voronoi diagram can be sequentially computed in $O(n \log n)$ time [29]. The best known parallel algorithm takes $O(\log n \log\log n)$ time with $n\frac{\log n}{\log\log n}$ processors, and $O(\log^2 n)$ time with $\frac{n}{/logn}$ processors [14]. The vertices of the Voronoi diagram are called *Voronoi vertices* and the points of the point set are referred as *Voronoi sites.* A Region of the Voronoi diagram is also called *face.*

In some cases it may not be possible to store both the point set and its Voronoi diagram due to memory limitations. This means we have to store only the point set and compute the Voronoi diagram whenever needed. But it is not very attractive

solution as computing Voronoi diagram is quite expensive. In this chapter we provide alternate solution to this problem; store only the Voronoi diagram and compute the point set when needed. We propose a fully parallel algorithm to compute the point set from the Voronoi diagram. The algorithm runs in $O(1)$ time with n processors on a CREW PRAM. This model allows simultaneous reads but does not allow simultaneous writes. As the Voronoi diagram G is a planar graph, we assume that the planar embedding of Voronoi diagram is given, i.e. for each Voronoi vertex v_j its adjacent vertices are given in counter-clockwise order. We are also given the co-ordinates of the Voronoi vertices. This definition of planar embedding is consistent with [12].

7.2 Computing the point set

The following properties of Voronoi diagram of a point set S are known [29] (it is assumed that points are in general position— no 4 points are co-circular).

1. Every Voronoi vertex is intersection of exactly three edges of the diagram, i.e. Voronoi diagram is regular of degree three
2. Every Voronoi vertex (v) is center of the circle(say $C(v)$ defined by three points of S
3. For a Voronoi vertex v, the circle $C(v)$ does not contain any other points of s

Observe that if the Voronoi diagram G is having only two faces, i.e. S has only two points, then the Voronoi diagram G is just a straight line l (Voronoi edge), and the point set S is obviously not unique (any pair of points for which l is a perpendicular bisector will do).

Consider the case when G has three faces, then G will be as shown below (see Figure 7.1); G has only one Voronoi vertex (v) and three Voronoi edges incident at v. Let us put the origin of the co-ordinate system at v. If $S = \{p, q, h\}$, then p, q and h should lie on a circle centred at v; let the radius of this circle be r. Further, let the coordinate of p, q and h, be $(r\cos\alpha, r\sin\alpha)$, $(r\cos\beta, r\sin\beta)$ and $(r\cos\gamma, r\sin\gamma)$ respectively. Let m_1, m_2 and m_3 be the slopes of Voronoi edges, i.e., the equations of line corresponding to Voronoi Edges are $y = m_1x$, $y = m_2x$, and $y = m_3x$. (See Figure 7.1). As each Voronoi Edge is perpendicular to the line joining two points of S,

$$\frac{\sin\alpha - \sin\beta}{\cos\alpha - \cos\beta} = \frac{-1}{m_1}$$

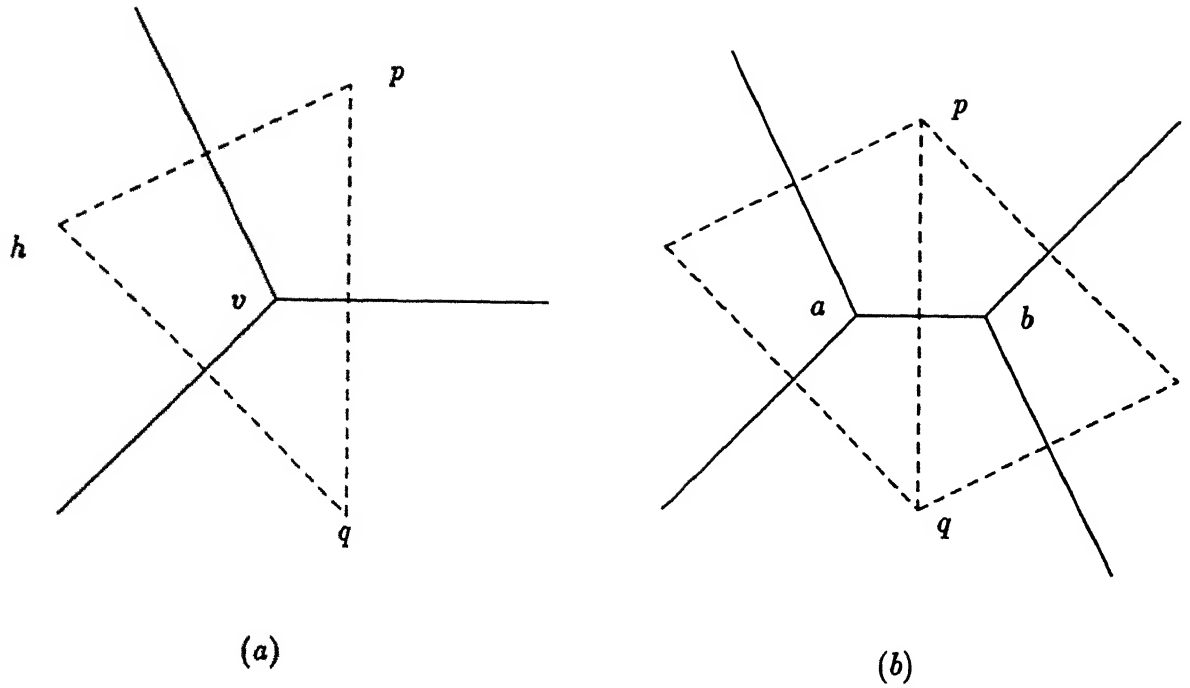


Figure 7.1: The Voronoi diagram

$$\frac{\sin\beta - \sin\gamma}{\cos\beta - \cos\gamma} = \frac{-1}{m_2}$$

and

$$\frac{\sin\gamma - \sin\alpha}{\cos\gamma - \cos\alpha} = \frac{-1}{m_3}$$

As each Voronoi edge is equidistant from (i.e. is a perpendicular bisector of) two points of S,

$$\sin\alpha + \sin\beta = m_1(\cos\alpha + \cos\beta)$$

$$\sin\beta + \sin\gamma = m_1(\cos\beta + \cos\gamma)$$

and

$$\sin\gamma + \sin\alpha = m_1(\cos\gamma + \cos\alpha)$$

These six equations can be solved to get α, β and γ. However, the radius r is not determined by these equations. Thus, when the number of Voronoi regions is three, then also the set S is not unique (see Figure7.1), as we can choose any r, but α, β and γ are fixed. Our main result is the following theorem:

Theorem 7.1 A Voronoi diagram G uniquely determines the point set S such that G is a Voronoi Diagram of S, provided that G has at least four regions, or equivalently S has at least four points.

Further co-ordinates of points in the set S can be obtained in $O(1)$ time with n processors on a Concurrent Read Exclusive Write (CREW) model.

Proof: Let us first consider the case when there are four Voronoi regions in G (see Figure 7.1); here there are two Voronoi vertices, and there is a Voronoi edge joining them. Let the two Voronoi vertices be a and b. Let p and q be the points in the regions adjacent to edge ab. Now with respect to Voronoi vertex a, we can determine the angles α, β and γ; hence points p and q are determined except for radius r. Similarly, we can also determine the angles of points p and q— α', β' and γ' with respect to Voronoi point b, also; again the radius r' has not been fixed by above equations. As p (and similarly q) should be the intersection point ray coming out of a and b (as angles are fixed, so are these rays), point p is unique and can be easily determined; similarly point q is unique and can be determined. Thus, the all points in set S can be uniquely determined from the Voronoi diagram G. In case, there are more than 4 regions, we can repeat the procedure for each pair of adjacent regions, and determine co-ordinates of all points.

As Voronoi diagram is a planar graph, number of edges e is of the same order as n, the number of Voronoi points. Thus, using $O(n)$ processors, we can then assign one processor to each Voronoi edge ab, and solve the equations in $O(1)$ time to determine the co-ordinates of 4 site points (each Voronoi point is defined by 3 points, here two points p and q are common). It is quite possible that co-ordinates of some point are determined more than once. At each Voronoi vertex we store the co-ordinates of the respective sites. So at each Voronoi point at most 3 Voronoi sites are stored. But each site is stored at many Voronoi vertices; a Voronoi site p is stored at all Voronoi vertices which belong to $V(p)$ (the Voronoi region of p). We show how to copy these Voronoi sites into an output array of size n, without concurrent writes. We use Euler tour technique [36]. For each Voronoi region ($V(p)$) we can find the edges that bound $V(p)$, using slight modification of Euler tour technique [7, 33], as follows.

for each edge (i,j) pardo next_on_face[(i,j)]:= next[reverse[(i,j)]] (=next[(j,i)]);

As input is given such that for each Voronoi vertex we know the Voronoi vertices adjacent to it, this algorithm can be applied. This gives a linked list (say $L(p)$) for each region ($V(p)$), such that the vertices of the linked list $L(p)$ are incident on the region $V(p)$. Now the head of the list $L(p)$ writes the co-ordinates of p in its slot. This avoids concurrent writes as only one processor writes in each location. This completes the proof of the theorem. ∎

Chapter 8

Concluding Remarks

The problem of finding tangents to a convex hull from a set of points considered in Chapter 2, seems to have some interesting applications. In this thesis the point set satisfies a special property. It needs to be explored whether this property can be relaxed. In particular, if we can find tangents from vertices of a star-shaped polygon to a convex hull inside the polygon in optimal $O(\log n)$ time, then it may be possible to obtain an optimal algorithm to find complete visibility polygon from a convex set [16].

In Chapter 3 an optimal parallel algorithm is described to compute the shortest path tree in a weak visible polygon. This algorithm does not use parallel triangulation algorithm and is optimal. It is still not known whether shortest path tree in simple polygon can be found without triangulating the polygon. Even all the sequential algorithms proposed first triangulate the polygon. To compute shortest path tree in a simple polygon optimally, either an optimal parallel algorithm for triangulation should be obtained, or a new method that does not need triangulation should be developed.

An optimal parallel algorithm is obtained for triangulating a star shaped polygon in Chapter 4. Triangulating any arbitrary polygon in optimal $O(\log n)$ time is still an open problem.

The convex hull (of a simple polygon) algorithm proposed in Chapter 5 is optimal. It has to be seen whether faster algorithms can be obtained.

Some fast parallel algorithms for computing visibility polygon are presented in Chapter 6. These algorithms work on a powerful CRCW COMMON PRAM. It has to be seen whether same bounds can be obtained on a weaker model. Finding a $O((\log\log n)^2)$ time algorithm with n processors would be more interesting. If one can

avoid the binary search bottle-neck, i.e. avoid binary search that is being performed to find intersection of visibility chains, then can be solved in desired resource bounds.

In Chapter 7 a fully optimal algorithm to get point set from a Voronoi diagram is described. It has to be seen whether this algorithm can be extended to the case, where points are not in general position.

References

[1] A. Aggarwal, B. Chazelle, L. Guibas, and C. Yap. Parallel computational geometry. *Algorithmica*, 3:293–327, 1988.

[2] S.G. Akl and K.A. Lyons. *Parallel Computational Geometry.* Prentice Hall, 1993.

[3] M.J. Atallah, D.Z. Chen, and H. Wagener. An optimal parallel algorithm for visibility of simple polygon from a point. *Journal of the ACM*, 38:516–533, 1991.

[4] M.J. Atallah, R. Cole, and M.T. Goodrich. Cascading divide and conquer: A technique for designing parallel algorithms. *SIAM Journal on Computing*, 18:499–532, 1989.

[5] M.J. Atallah and M.T. Goodrich. Efficient parallel solutions to some geometric problems. *Journal of Parallel and Distributed Computing*, 3:492–507, 1986.

[6] D. Avis and G. Toussiant. An optimal algorithm for determining visibility of simple polygon from an edge. *IEEE Transactions on Computers*, C-30:910–914, 1981.

[7] B.G. Baumgart. A polygon representation for computer vision. In *AFIPS conference proceedings*, pages 589–596, 1975.

[8] O. Berkman, B. Schieber, and U. Vishkin. Optimally doubly logarithmic algorithms for finding all nearest smaller values. *Journal of Algorithms*, 14:344–370, 1993.

[9] O. Berkman and U. Vishkin. Recursive star parallel data structure. *SIAM Journal on Computing*, 22:221–242, 1993.

[10] V. Chandru, S.K. Ghosh, A. Maheshwari, V.T. Rajan, and S. Saluja. Nc algorithms for minimum link path and related problems. Technical Report CS-90/3, TIFR, 1991.

[11] B. Chazelle. Triangulating simple polygon in linear time. *Discrete and Computational Geometry*, 6:485–524, 1991.

[12] N. Chiba, T. Nishizeki, S. Abe, and T. Ozawa. A linear algorithm for embedding

planar graphs using pq-trees. *Journal of computer and system sciences*, 30:54–76, 1985.

[13] R. Cole and M.T. Goodrich. Optimal algorithms for point set and polygon problems. *Algorithmica*, 7:3–23, 1992.

[14] R. Cole, M.T. Goodrich, and C. Dunlaing. Merging free tree in parallel for efficient Voronoi diagram construction. In *LNCS, 443*, pages 432–445, 1990.

[15] A. Fourier and D.Y. Montuno. Triangulating simple polygons and equivalent problems. *ACM Transactions on Graphics*, 3:153–174, 1984.

[16] S.K. Ghosh. Computing visibility polygon from a convex set and related problems. *Journal of Algorithms*, 12:75–95, 1991.

[17] S.K. Ghosh and A. Maheshwari. Optimal parallel algorithm for determining intersection type of two star shaped polygons. Technical Report 163, TIFR, 1990.

[18] S.K. Ghosh, A. Maheshwari, S.P. Pal, S. Saluja, and C.E. Veni Madhaven. Characterizing weak visibility polygons and related problems. Technical Report IISc-CSA-90-01, IISc, Bangalore, 1990.

[19] A. Gibbons and P. Spirakis. *Lectures on Parallel Computation.* Cambridge University Press, 1993.

[20] M.T. Goodrich. Finding convex hull of a sorted point set in parallel. *Information Processing Letters*, 26:173–179, 1987.

[21] M.T. Goodrich. Triangulating a polygon in parallel. *Journal of Algorithms*, 10:327–351, 1989.

[22] M.T. Goodrich, B. Shauck, and S. Guha. Parallel methods for visibility and shortest path problems in simple polygons. *Algorithmica*, 8:461–486, 1992.

[23] L. Guibas, J. Hershberger, D. Leven, M. Sharir, and R.E. Tarjan. Linear time algorithms for visibility and shortest path problems inside simple polygons. In *Proceedings of 2nd annual ACM Symposium on Computational Geometry*, pages 1–13, 1986.

[24] D.G Kirkpatrick. Optimal search in planar subdivisions. *SIAM Journal on Computing*, 12:23–35, 1983.

[25] D.T. Lee. Visibility of a simple polygon. *Computer Vision Graphics and Image Processing*, 22:207–221, 1983.

[26] D.T. Lee and F.P. Preparata. Eucludian shortest paths in the presence of rectilinear barriers. *Networks*, 14:393–410, 1984.

[27] K. Mehlhorn. *Data Structures and Algorithms 3: Multidimensional Search and Computational Geometry.* Springer-Verlag, Berlin, 1984.

[28] R. Miller and Q.F. Stout. Efficient parallel convex hull algorithms. *IEEE Transactions on Computers*, C-37:1605–1618, 1988.

[29] F.P. Preparata and M.I. Shamos. *Computational Geometry: An Introduction.* Springer-Verlag, New York, 1985.

[30] P. Ragde. The parallel simplicity of compaction and chaining. *Journal of Algorithms*, 14:371–380, 1993.

[31] J.R. Sack and S. Suri. An optimal algorithm for detecting weak visibility of a polygon. *IEEE Transactions on Computers*, C-39:1213–1219, 1990.

[32] S. Saxena. *Class Notes CS698b: 1993-94 1st sem.*

[33] S. Saxena. *Design and analysis of combinatorial and computational geometry problems for parallel execution.* PhD thesis, IIT, Delhi, 1989.

[34] S. Suri. A linear time algorithm for minimum link path in a simple polygon. *Computer Vision Graphics and Image Processing*, 35:99–110, 1986.

[35] R.E. Tarjan and C. Van Wyk. An o($\log \log n$) algorithm for triangulating a simple polygon. *SIAM Journal on Computing*, 17:143–178, 1988.

[36] R.E. Tarjan and U. Vishkin. An efficient parallel biconnectivity algorithm. *SIAM Journal on computing*, 14:862–874, 1985.

[37] H. Wagener. Optimally parallel algorithms for convex hull determination. Manuscript, Technical University of Berlin, 1985

[38] C.K. Yap. Parallel triangulation of a polygon in two calls to trapezoidal map. *Algorithmica*, 3:279–288, 1988.